GitHub

目錄

前言 0

1. 探索GitHub 1

    1.1. 什么是GitHub 1.1

    1.2. GitHub亮点 1.2

    1.3. 探索GitHub 1.3

2. 加入GitHub 2

    2.1. 创建GitHub账号 2.1

    2.2. 浏览托管项目 2.2

    2.3. 社交网络 2.3

3. 项目托管 3

    3.1. 创建新项目 3.1

    3.2. 操作版本库 3.2

    3.3. 公钥认证管理 3.3

    3.4. 版本库钩子扩展 3.4

    3.5. 建立主页 3.5

4. 工作协同 4

    4.1. Fork + Pull模式 4.1

    4.2. 共享版本库 4.2

    4.3. 组织和团队 4.3

    4.4. 代码评注 4.4

    4.5. 缺陷跟踪 4.5

    4.6. 维基 4.6

5. 付费服务 5

    5.1. GitHub收费方案 5.1

    5.2. GitHub企业版 5.2

6. GitHub副产品 6

    6.1. GitHub:Gist 6.1

    6.2. 其他版本控制工具支持 6.2

        6.2.1. 用SVN操作GitHub 6.2.1

        6.2.2. 用Hg操作GitHub 6.2.2

- 6.3. 客户端工具 6.3
- 6.4. 其他 6.4
- 7. 附录：轻量级标记语言 7
- 贡献者列表 8

前言

> 来源：http://www.worldhello.net/gotgithub/index.html 作者：蒋鑫

动笔写GitHub不是因为我对其了解，恰恰是对其太不了解。

在我的《Git权威指南》 [1] 一书中，涉及到GitHub的只有区区三页纸，这显然回答不了读者对于GitHub的诸多疑问。 记得在《Git权威指南》刚刚完稿之际，机械工业出版社华章公司的杨福川编辑就鼓动我写一本关于GitHub的书，我用了好多理由推辞了。 头条理由就是我真的累着了。在每一章节开始动笔之时，都好像是坐在了中学语文考试的考堂上写作文，时间快到了可仍然动不了笔， 再写一本书无疑要重复这一痛苦的经历。 第二个理由是我更喜欢编程，而不是写文档，尤其写GitHub会有大量截图、图像处理的琐碎工作。 第三个理由彻底让编辑投降，那就是GitHub是一个国外网站，也许书一出，【此句已被原作者删除】。

让我最终决定动笔，是源于CSDN蒋总在美国拜访GitHub总部后告诉我的一些见闻，我对GitHub如此成功运作产生了兴趣，于是开始研究GitHub的博客，愈发发现GitHub是一群有趣的人在做的有趣的事，如果只把GitHub当作一个Git服务器，实在是暴殄天物。GitHub已经并将继续获得成功，若真能凭借此书把GitHub尽量全面地展现，让每一个Git使用者用好GitHub也是一件幸事。

这本书将采用GitHub的方式进行撰写和发布 [2] ，任何人都可以看到本书（包括源码），更可以用GitHub的方法参与本书的撰写和纠错。网络出版对于我和杨福川编辑都是一个全新的体验。感谢Git，让我在一年内拥有了两种不同的出版体验。

– 蒋鑫, 2011.12

1. 探索GitHub

熟悉Git的人几乎都知道并喜欢GitHub，反过来GitHub也吸引更多的人来使用Git。GitHub正在成为开源项目托管的主要平台，是什么成就了GitHub？

> 也可以参考：
>
> 本书并非一本介绍Git的书，并且假设读者已经掌握了Git的相关操作。如果读者对Git尚不了解，可以参考我写的 [《Git权威指南》] [注1]一书。此外还可以从网上找到很多免费的、很好的Git资料，如：Git社区书、Pro Git等。

[注1]: ISBN：9787111349679, 由机械工业出版社华章公司于2011年7月出版。

1.1. 什么是GitHub

GitHub（网址 https://github.com/）是一个面向开源及私有软件项目的托管平台，因为只支持Git作为唯一的版本库格式进行托管，故名GitHub。

GitHub的注册用户已经超过百万[1]，托管的版本库数量已超三百万，其中不乏知名的开源项目，如：Ruby on Rails[2]、Hibernate[3]、phpBB[4]、jQuery[5]、Prototype[6]、Homebrew[7]等。

GitHub于2008年4月10日正式发布[8]，相比始于1999年的SourceForge[9]和2005年的GoogleCode[10]，GitHub后来者居上。以2011年的数据从代码提交数量上看，GitHub已经超越其前辈[11]，如图1-1所示。

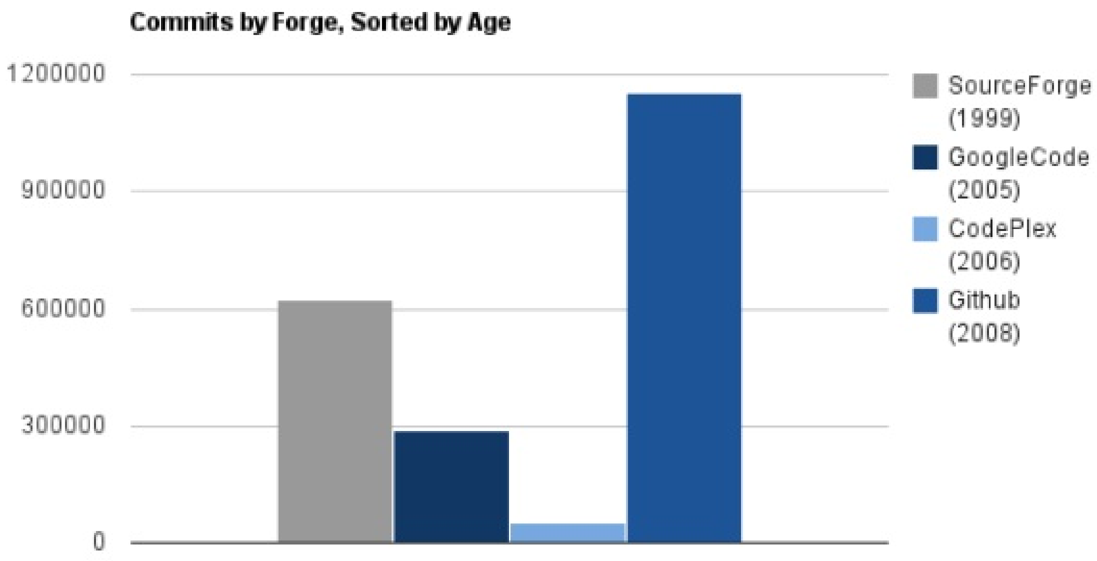


图1-1：开源项目托管平台提交数量对照

对于一个开源项目，从开发角度讲大体上分为两类人群，一类称为核心开发团队，他们可以向保存源代码的版本库提交，即对源代码的修改具有最终的决定权。另外一类称为贡献者，他们不属于核心开发团队，虽然也能看到源代码，但无权向版本库提交修改。

采用传统的集中式版本控制系统（如SVN）的开源项目，这两个群体的用户体验都不是太好。如图1-2所示，项目的贡献者（非核心成员）很不“高兴”，因为他们即便有修改源代码的能力和渴望，也不能直接向版本库提交，要想成为提交者需要一个很长的建立信任的过程。然而即便是核心开发团队的成员，体验也不是太好，因为凡是涉及到版本库的操作（检入、检出、查看日志等）都需要在联网的状态下进行，网络带宽对用户体验影响相当大。

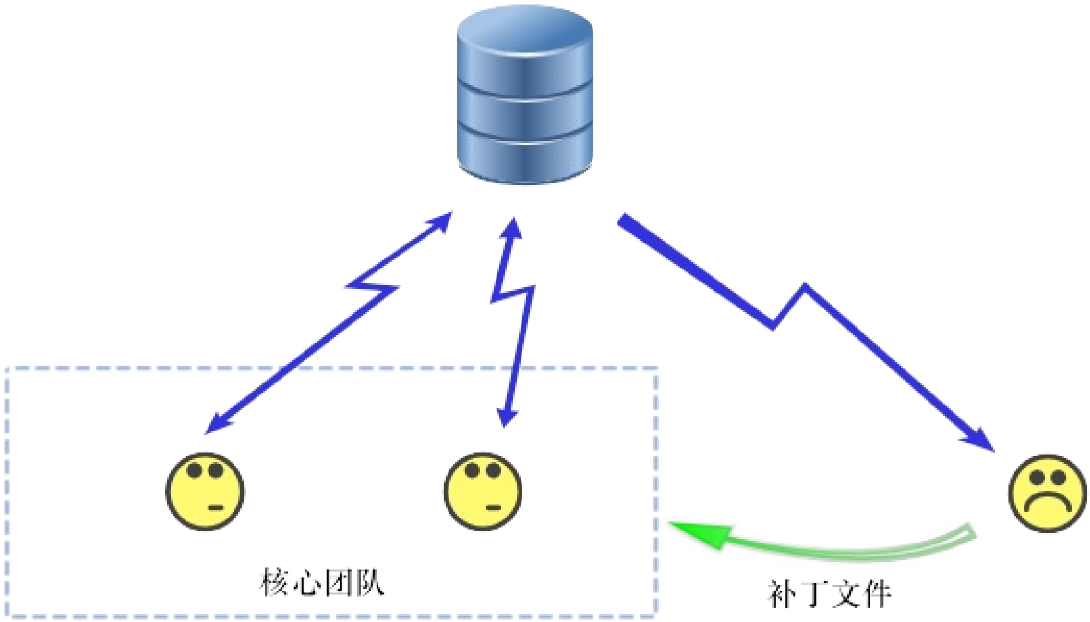


图1-2：使用集中式版本控制系统

Git等分布式版本控制系统的出现，彻底颠覆了原有代码管理的组织模式。使用Git，不再依赖唯一的、集中式的版本库，而是每个开发者本地都拥有一份完整的版本库。Git并不排斥集中式的使用模式，但更倾向于将集中式版本库称为共享版本库。核心开发团队的成员和贡献者（非核心成员）都可以从共享版本库克隆一份本地版本库，但只有核心团队成员才可以将自己本地版本库的提交推送到共享版本库上。如图1-3所示。

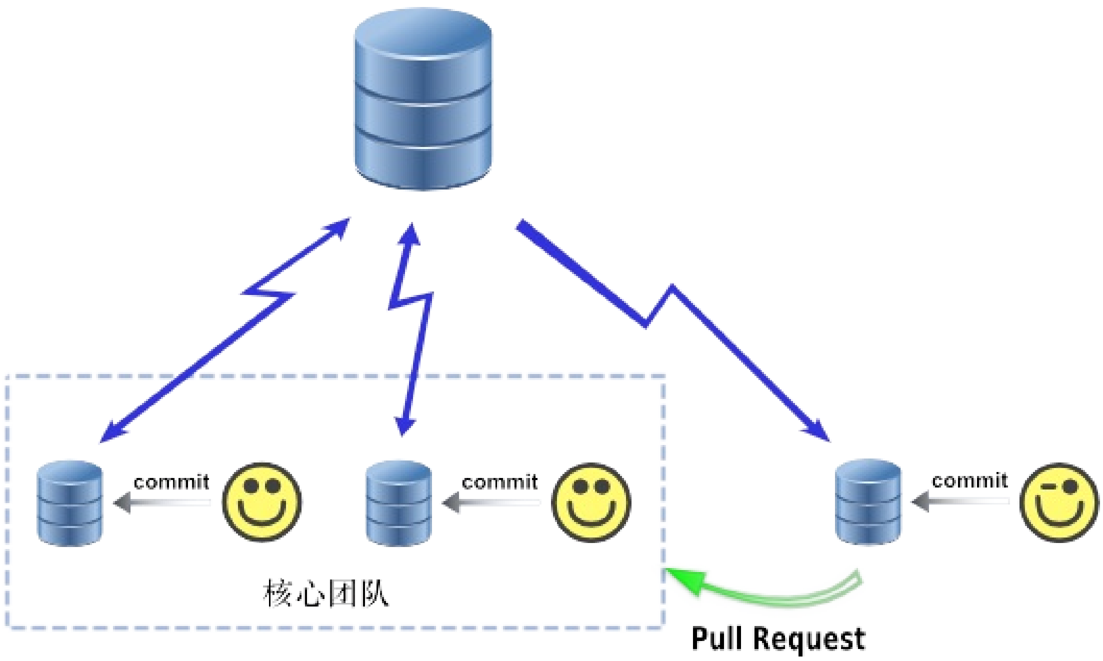


图1-3：使用分布式版本控制系统

使用Git做版本控制（如图1-3所示），核心开发团队非常“高兴”，因为他们和共享版本库之间不必一直保持连接状态，诸如查看日志、提交、创建分支等几乎全部操作都（脱离网络）在本地的版本库中完成。项目贡献者（非核心成员）也不再那么沮丧，因为版本库人人皆可更改（当然是对本地版本库而言）。稍微让贡献者感到困难的就是如何将自己对项目的改进被核心开发团队所了解并接纳。Git提供了多种途径，一个方法是先用git format-patch命令将本地提交转换为补丁文件或补丁文件序列，再通过邮件发送给核心开发团队。另外一个办法就是搭建一个自己专有的共享版本库，通过邮件创建一个拉拽请求（Pull Request），让核心团队的开发者到自己的版本库来抓取（Pull）。

GitHub的出现进一步推动了Git的普及，简化了版本控制的管理和操作流程，为开发者提供了更好的交流平台。如图1-4所示。

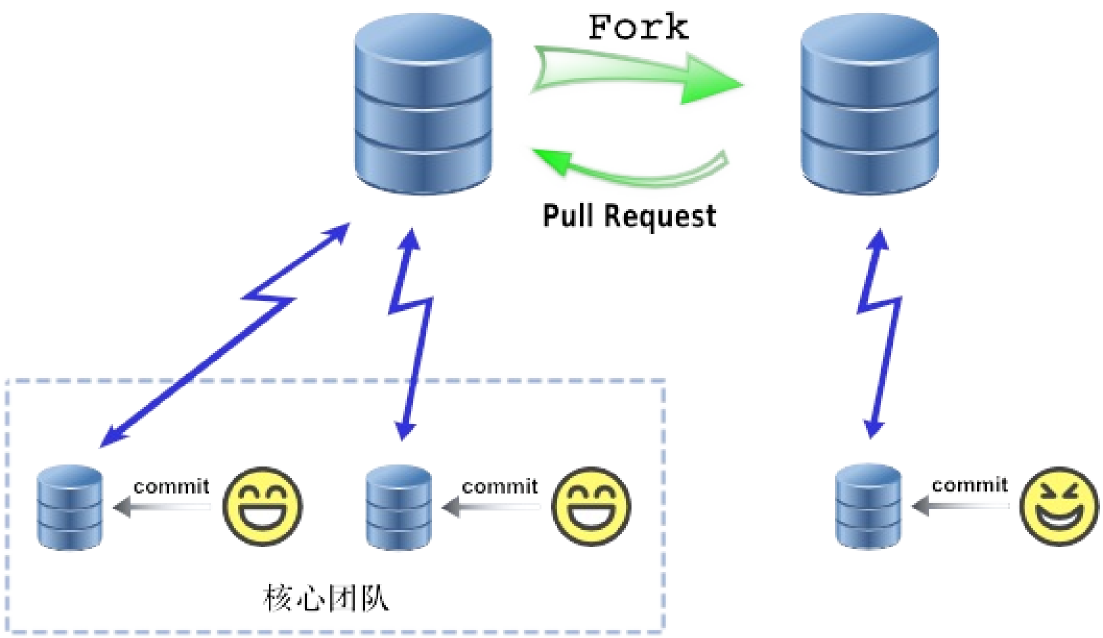


图1-4：GitHub的协同模式

使用GitHub，无论是项目的核心开发团队，还是普通的项目贡献者都工作得非常“愉快”。创建项目变得非常轻松，创建者只需在GitHub上点击一下鼠标即可创建一个新版本库，通过简单的Web操作即可完成项目授权进而组建项目核心团队。在GitHub中，非核心团队成员参与项目也很容易。先找到自己希望参与的项目，然后只需在Web上点击一下鼠标即可在自己的托管空间下创建一个派生（fork）的项目，并对派生项目的版本库具有读写的完全权限，就好像这个项目原本就是由自己创立的那样。当贡献者完成开发并向自己派生的版本库推送后，可以通过GitHub的Web界面向项目的核心开发团队发送一个Pull Request，请求审核。项目的核心团队收到Pull Request后审核代码，审核通过后可以直接通过Web界面执行合并操作接纳贡献者的提交。

1.2. GitHub亮点

是什么让GitHub如此成功？GitHub有什么魔力？

1. 只用Git。

   GitHub只支持Git格式的版本库托管，而不像其他开源项目托管平台还对CVS、SVN、Hg等格式的版本库进行托管。GitHub的哲学很简单，既然Git是最好的版本控制系统之一（对于很多喜欢Git和GitHub的人没有之一），没有必要为兼顾其他版本控制系统而牺牲Git某些独有特性。因此没有支持其他版本控制系统的历史负担，是GitHub成功的要素之一。

   只用Git并不是说GitHub完全无视其他版本控制系统的使用者，相反，GitHub面向SVN（Subversion）用户和Hg（Mercurial）用户开发了接口，让这些用户可以使用SVN或Hg的客户端工具访问Git版本库。

2. 对Git的完整支持。

   相比其他开源项目托管平台，GitHub对Git版本库提供了完整的协议支持，支持HTTP智能协议、Git-daemon、SSH协议。相比只支持HTTP协议的GoogleCode，GitHub通过SSH协议可以实现版本库访问的无口令认证（实际上使用HTTP协议也可以免口令输入。即通过文件 `~/.netrc` 写入HTTP认证的明文口令。具体文件格式参见ftp命令MAN手册中相关介绍）。

3. 无处不在的Git。

   除了在版本库托管上使用Git，Git还被GitHub应用到更多领域。维基使用Git，可以通过克隆维基所在的版本库，离线修改维基；在线粘贴数据的Gist网站[2]使用Git，记录变更历史；以及在Jekyll应用的帮助下，用Git版本库维护个人网站和博客等。

4. 在线编辑文件。

   GitHub提供了在线编辑文件的功能，不熟悉Git的用户也可以直接通过浏览器修改版本库里的文件。

5. 社交编程。

   将社交网络引入项目托管平台是GitHub的创举。用户可以关注项目、关注其他用户进而了解项目和开发者动态。项目的派生（Fork）和拉拽请求（Pull Request）构成GitHub最独具一格的工作模式。对提交代码的逐行评注及Pull Request构成了GitHub特色的代码审核。

6. 商业上的成功。

GitHub通过私有版本库托管、面向企业的版本库托管和项目管理平台、人员招聘等付费服务获得了商业上的成功，这种成功使得GitHub不必以页面中嵌入广告的方式维持运营，最大的受益者还是用户。

7. 关注细节。

   GitHub网站采用了Ruby on Rails架构，在Web设计中运用了大量的JavaScript、AJAX、HTML5等技术，支持对使用Markdown等标记语言的内容进行渲染和显示等。关注细节使得GitHub成为了项目托管领域的后起之秀。

1.3. 探索GitHub

打开浏览器，访问网址 https://github.com/，来探索GitHub吧。GitHub的首页（图1-5所示）特意给出了Git和GitHub的音标，可能不少国人需要据此校准一下Git的读音(《Git权威指南》第1页就提到了两种常见的对Git的读音错误。)。

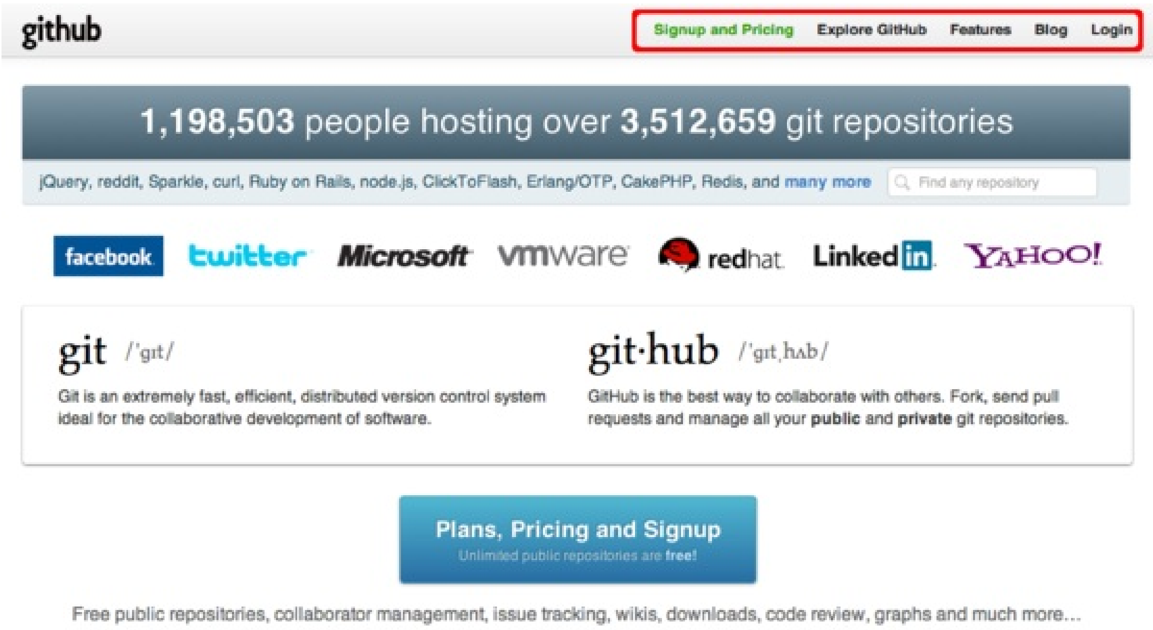


图1-5：GitHub的首页

在首页的右上角是导航条，从左至右分别是：注册和收费方案、探索GitHub、功能、博客和登录。还醒目地显示出不断增长着的注册用户数和托管的版本库数目。

如果想要了解GitHub上哪些项目最热门，进而寻找到好的开源产品，那么可以从导航条中的“Explore GitHub”开始。图1-6显示通过对社交数据的分析得到的托管版本库动态趋势。



图1-6：版本库动态趋势

还可以根据感兴趣的人数、建立分支的数量、关注程度等寻找热门项目。图1-7显示分支最多的项目是Homebrew —— 一款用ruby开发的苹果Mac OS X通用的非官方包管理软件。考虑到不断攀升的苹果用户数量以及易于上手的ruby语言，这并不奇怪。



图1-7：热门版本库排行

图1-8显示了托管版本库所用编程语言的动态分布，程序员多掌握几个热门编程语言一定会对找工作有帮助。;-)

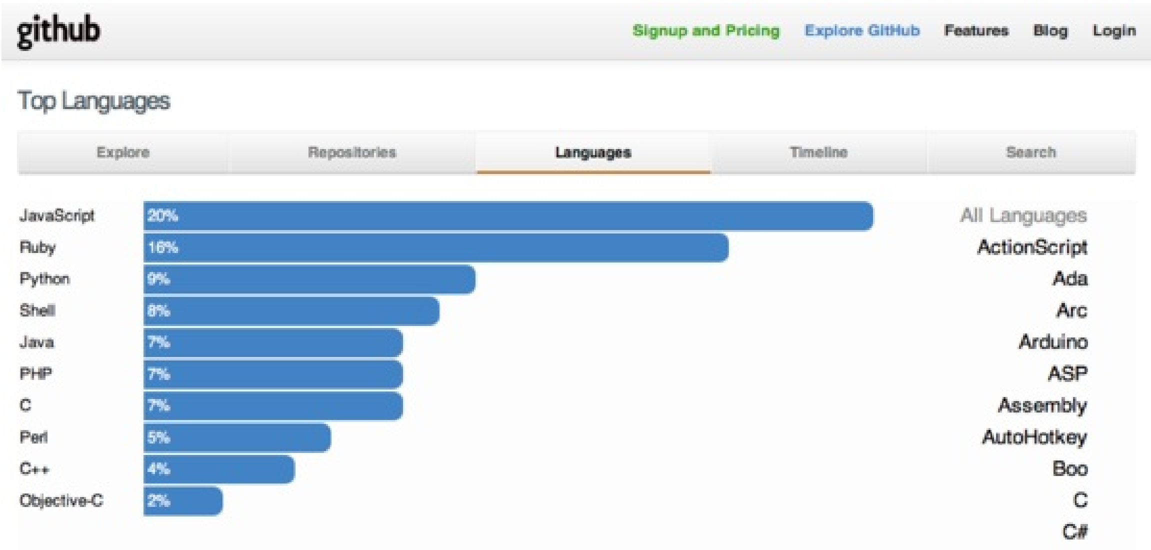


图1-8：托管项目的编程语言统计

GitHub通过屏幕截图等方式介绍了GitHub的常见功能，可以通过点击导航条中的“Features”访问到。如图1-9可以看到在项目管理中，如何利用GitHub提供的团队管理功能、维基、缺陷追踪以及代码审核。

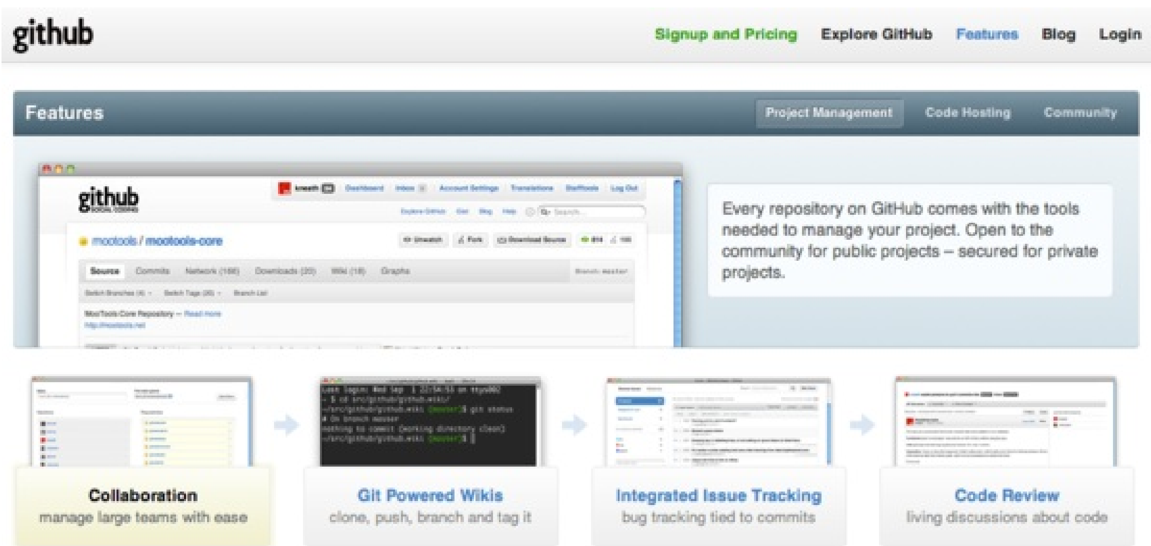


图1-9：GitHub功能介绍

博客也是了解GitHub的一个重要的途径，可以获知GitHub的最新动态，如最新改进等。图1-10显示的是GitHub在感恩节推出的促销活动：收费服务免费试用一个月！[1]如果及时关注博客就不会错过噢。

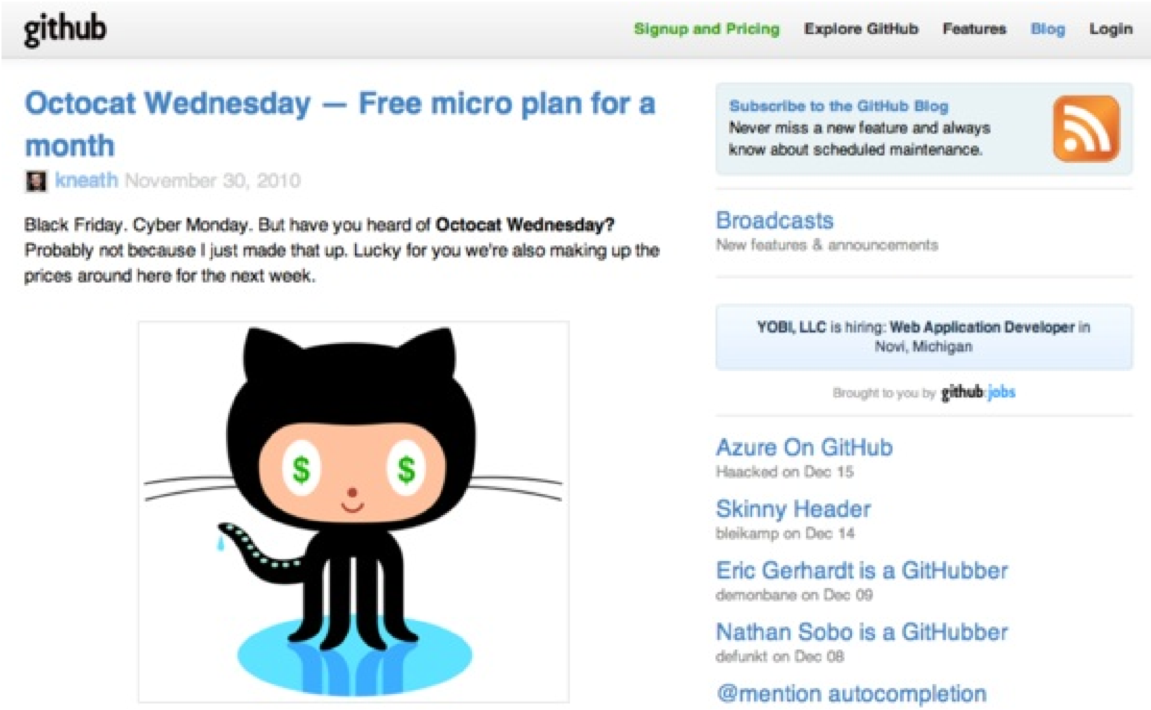


图1-10：GitHub博客

图1-10的博客中一个由小章鱼和小猫组合而成的吉祥物，名字叫做Octocat。这个可爱的GitHub吉祥物时不时会出来带给你惊喜。

马上到GitHub上注册，开始GitHub之旅。

2. 加入GitHub

本章介绍如何在GitHub上注册账号，并以现有项目为例介绍GitHub的主要功能。

2.1. 创建GitHub账号

注册GitHub账号，只要点击导航条中的“Signup and Pricing”，或者点击首页中那个大大的“Plans, Pricing and Signup”按钮，即进入收费方案介绍及注册页面。

收费？不必担心，开源软件托管是GitHub的基石，对于开源项目的版本库（即非私有版本库）的托管，GitHub是免费的。在收费方案及注册页面中，最上面的就是针对于开源的免费托管方案，如图2-1所示。

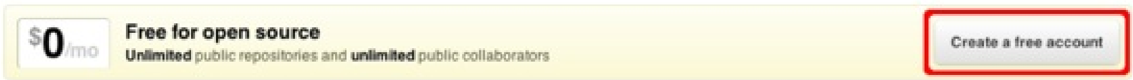


图2-1：针对开源项目（公开版本库）的免费方案

至于本页其他付费方案，将在后面的章节介绍。点击免费方案右侧的“Create a free account”按钮，就进入到注册页面，如图2-2所示。

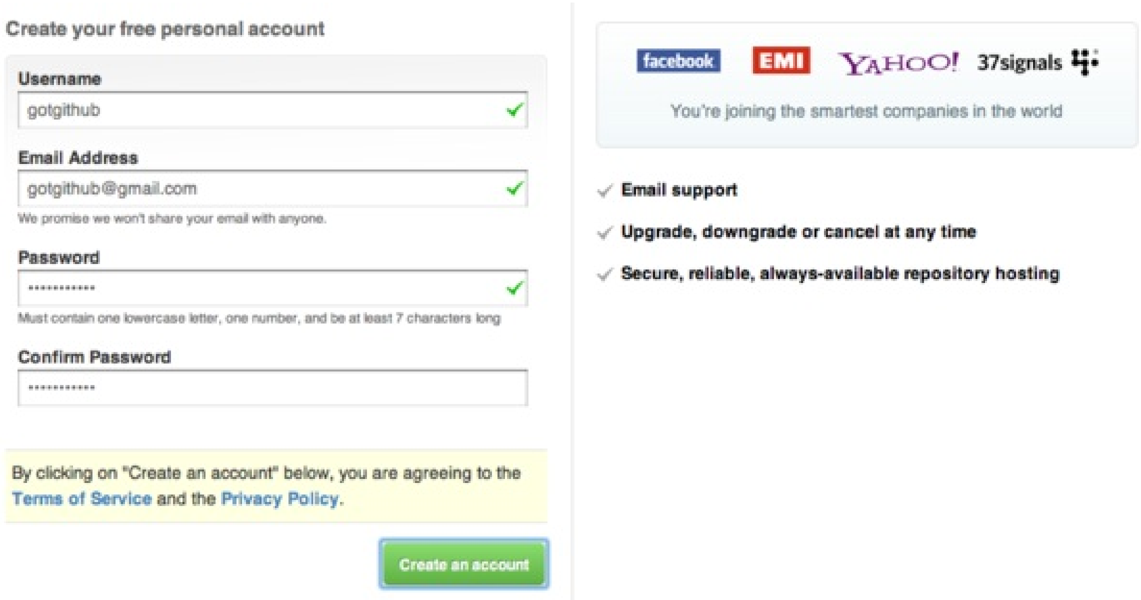


图2-2：账号注册

GitHub的注册页面非常简洁，只有登录ID，邮件地址和口令需要输入。要注意的是每个邮件地址只能注册一次。

注册完毕即以新注册的账号自动登录。登录后即进入用户的仪表板（Dashborad）页面。首次进入的仪表板页面还会在其中显示GitHub BootCamp（GitHub 新手训练营）的链接，以帮助新用户快速入门。如果图2-3所示。

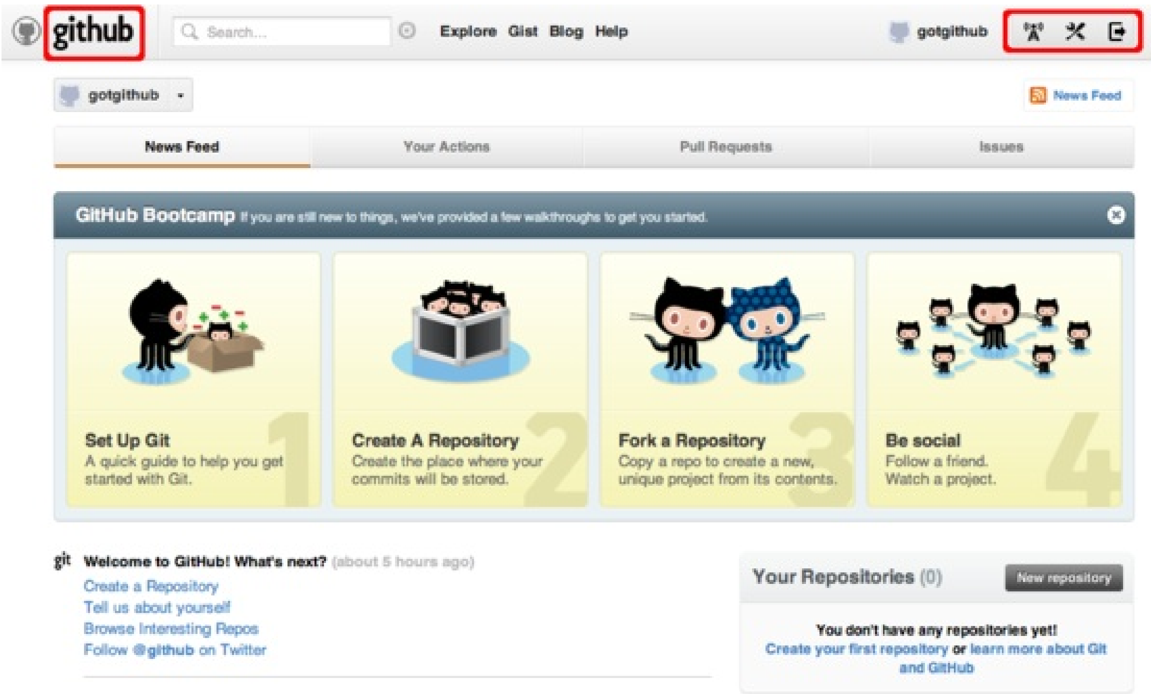


图2-3：登录后的GitHub首页

仪表板页面是用户最重要的页面，因为创建新项目（新版本库）的链接就位于该页面。重新设计的GitHub用户界面[1]中跳转到仪表板页面的链接不像之前那么直观，鼠标移动到页面左上角的“github”文字图标会发现此图标可以点击，该文字图标即是进入仪表板页面的快捷。

在页面右上方显示当前登录用户的名称和头像。图2-3中显示登录用户为 gotgithub，而用户头像因为尚未设置所以显示为缺省图片——GitHub吉祥物Octocat的剪影。在页面右上方还有三个图标，从左至右分别是：通知、账号设置和退出。点击账号设置图标对账号进行进一步设置，如图2-4所示。

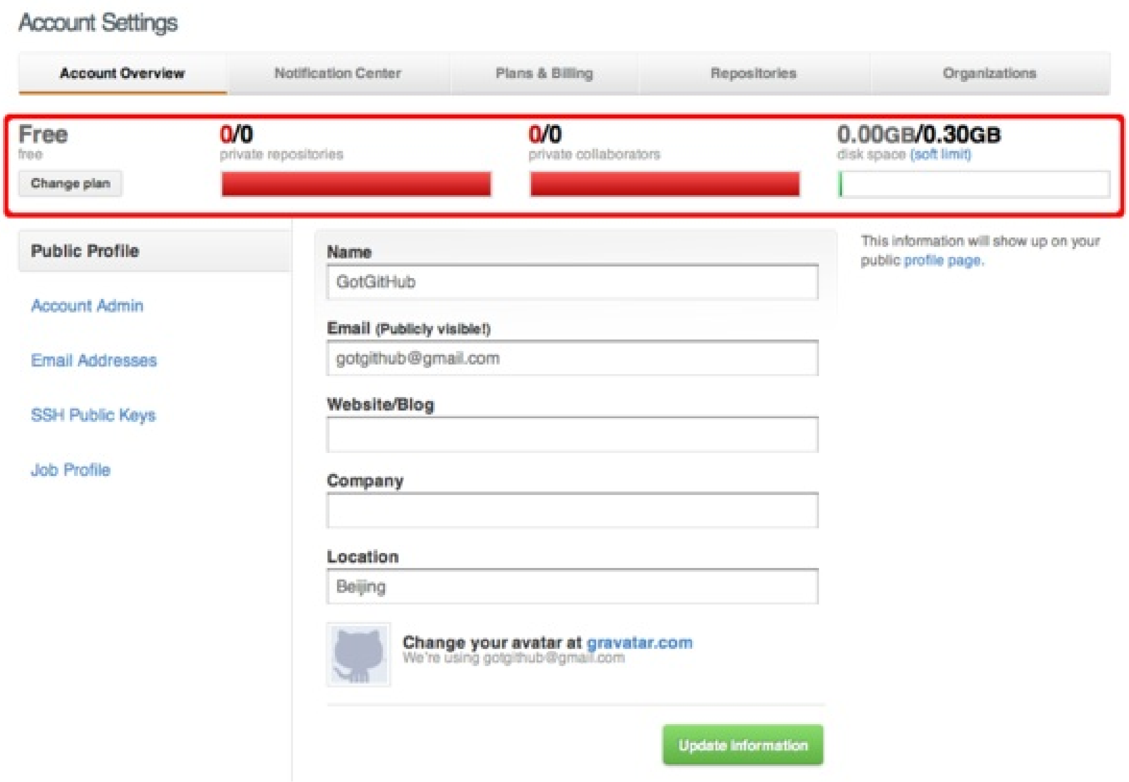


图2-4：账户设置页

账号设置的第一个页面是对用户公开身份信息进行设置，所有内容均为可选项，如果填写将显示在个人页面中，并能被所有人访问。注意修改用户头像需要访问第三方头像设置网站：gravatar.com，Gravatar网站提供的头像服务是一个通用服务，可为大部分Web应用所使用。

图2-4中还显示了当前用户使用的GitHub托管方案（Free）和使用统计。因为当前注册用户选择的是免费方案，所以可用的私有版本库数量和私有空间的协同者数目都是零。免费方案拥有300MB托管空间，因当前尚未创建版本库托管，所以空间占用为零。GitHub对开源软件的300MB托管空间限制并非硬性限制，可以申请扩增托管空间，如果不存在滥用情况的话。

点击菜单中的“Account Admin”，可以更改口令、查看API Token、修改用户名，以及删除自身账号，如图2-5所示。

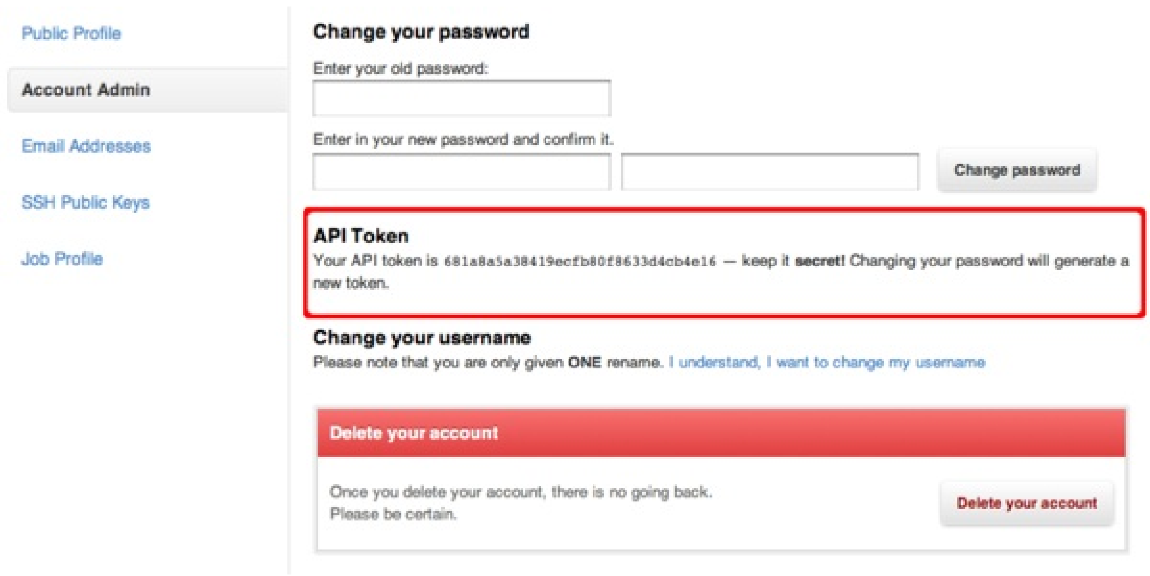


图2-5：账户管理

其中API Token是和用户口令相关的密钥，当用户口令更改时API Token也随之更改。GitHub的某些应用会使用API Token进行身份认证，从而避免直接使用用户口令造成泄露的风险。API Token若泄露的危害要远远小于口令泄露，这因为API Token不能用于登录GitHub网站等，而且一旦API Token泄露可以很容易通过更改口令的方式更换API Token。

点击菜单中的“Email Addresses”，可以添加和删除邮件地址，如图2-6所示。GitHub允许为一个账号绑定多个邮件地址，以便能够将Git版本库中的提交（提交者以“用户名 ”的格式给出）正确对应到GitHub账户。

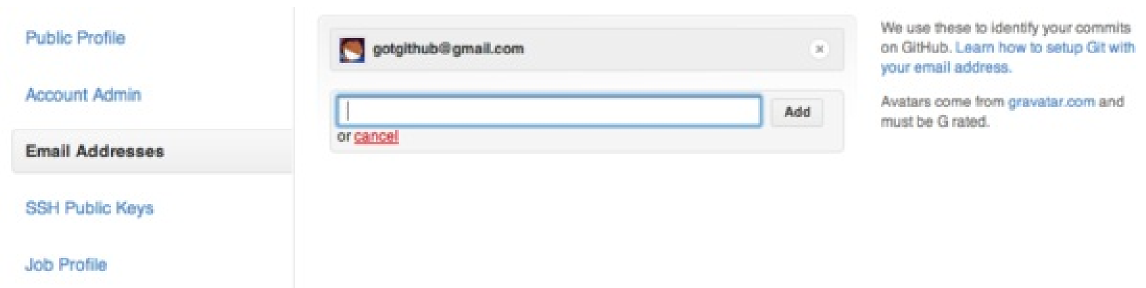


图2-6：邮件地址管理

GitHub为托管的Git版本库提供SSH协议支持，即用户可以用公钥认证的方式连接到GitHub的SSH服务器。下面的示例用ssh命令连接github.com的SSH服务，登录用户名为git（所有GitHub用户共享此SSH用户名，不要写成其他）。

```
$ ssh -T git@github.com
Permission denied (publickey).
```

上面的示例显示登录失败，这是因为我们尚未在GitHub账户中正确设置公钥认证。图2-7显示的是GitHub的SSH公钥设置界面。

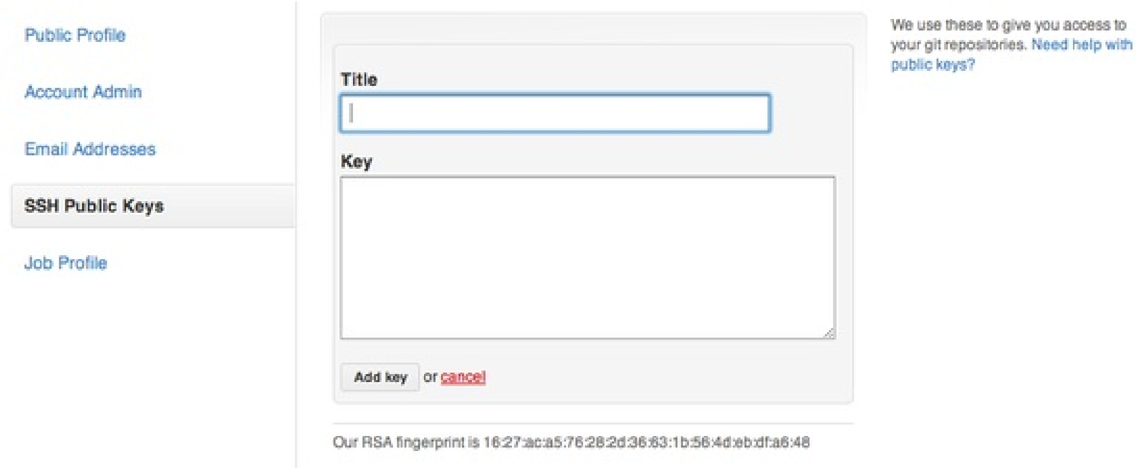


图2-7：SSH公钥管理

要想向GitHub添加SSH公钥，首先要确保正确生成了对应的公钥/私钥对。关于SSH公钥认证，在我的《Git权威指南》一书的“第29章使用SSH协议”中有详细介绍，这里仅做简要的介绍。

GitHub的SSH服务支持OpenSSH格式的公钥认证，可以通过Linux、Mac OS X、或Cygwin下的ssh-keygen命令创建公钥/私钥对。命令如下：

```
$ ssh-keygen
```

然后根据提示在用户主目录下的.ssh目录中创建默认的公钥/私钥对文件，其中 `~/.ssh/id_rsa` 是私钥文件， `~/.ssh/id_rsa.pub` 是公钥文件。注意私钥文件要严加保护，不能泄露给任何人。如果在执行ssh-keygen命令时选择了使用口令保护私钥，私钥文件是经过加密的。至于公钥文件 `~/.ssh/id_rsa.pub` 则可以放心地公开给他人。

也可以用ssh-keygen命令以不同的名称创建多个公钥，当拥有多个GitHub账号时，非常重要。这是因为虽然一个GitHub账号允许使用多个不同的SSH公钥，但反过来，一个SSH公钥只能对应于一个GitHub账号。下面的命令在 `~/.ssh` 目录下创建名为gotgithub的私钥和名为gotgithub.pub的公钥文件。

```
$ ssh-keygen -C "gotgithub@gmail.com" -f ~/.ssh/gotgithub
```

当生成的公钥/私钥对不在缺省位置（~/.ssh/id_rsa等）时，使用ssh命令连接远程主机时需要使用参数-i 指定公钥/私钥对。或者在配置文件~/.ssh/config中针对相应主机进行设定。例如对于上例创建了非缺省公钥/私钥对~/.ssh/gotgithub，可以在~/.ssh/config配置文件中写入如下配置。

```
Host github.com
  User git
  Hostname github.com
  PreferredAuthentications publickey
  IdentityFile ~/.ssh/gotgithub
```

好了，有了上面的准备，就将~/.ssh/gotgithub.pub文件内容拷贝到剪切板。公钥是一行长长的字符串，若用编辑器打开公钥文件会折行显示，注意拷贝时切莫在其中插入多余的换行符、空格等，否则在公钥认证过程因为服务器端和客户端公钥不匹配而导致认证失败。命令行下可直接用pbcopy命令[2]将文件内容拷贝到剪切板以避免拷贝错误：

```
$ cat ~/.ssh/gotgithub.pub | pbcopy
```

> Mac下的命令行工具pbcopy和pbpaste可以在命令行下操作剪贴板，Linux下的命令行工具xsel亦可实现类似功能。在Linux下可以创建别名用xsel命令来模拟pbcopy和pbpaste。 alias pbcopy='xsel --input' alias pbpaste='xsel --output'

然后将公钥文件中的内容粘贴到GitHub的SSH公钥管理的对话框中，如图2-8所示。

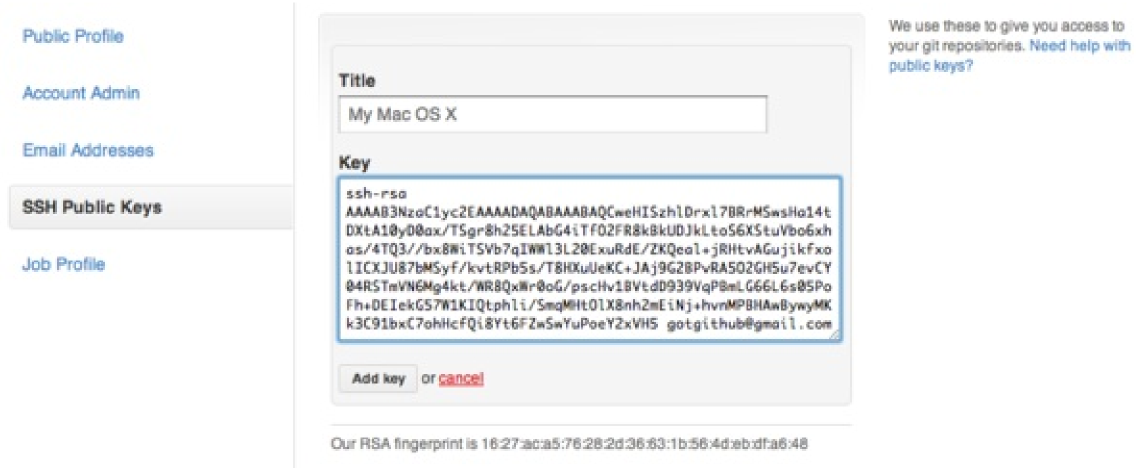


图2-8：添加SSH公钥认证

设置成功后，再用ssh命令访问GitHub，会显示一条认证成功信息并退出。在认证成功的信息中还会显示该公钥对应的用户名。

```
$ ssh -T git@github.com
Hi gotgithub! You've successfully authenticated, but GitHub does not provide shell access
```

如果您未能看到类似的成功信息，可以通过在ssh命令后面添加-v参数加以诊断，会在冗长的会话中看到认证所使用的公钥文件等信息。然后比对所使用的公钥内容是否和GitHub账号中设置的相一致。

```
$ ssh -Tv git@github.com
...
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
debug1: Offering RSA public key: /Users/jiangxin/.ssh/gotgithub
...
debug1: Entering interactive session.
Hi gotgithub! You've successfully authenticated, but GitHub does not provide shell access
...
```

账号设置的最后一项是向GitHub提供你的求职信息。GitHub作为一个优秀程序员的聚集地，已成为一个IT人才招聘的途径，如果你需要找工作的话，提供简历并打开“Available for hire”选项，GitHub会向你推荐合适的工作机会。如图2-9所示。

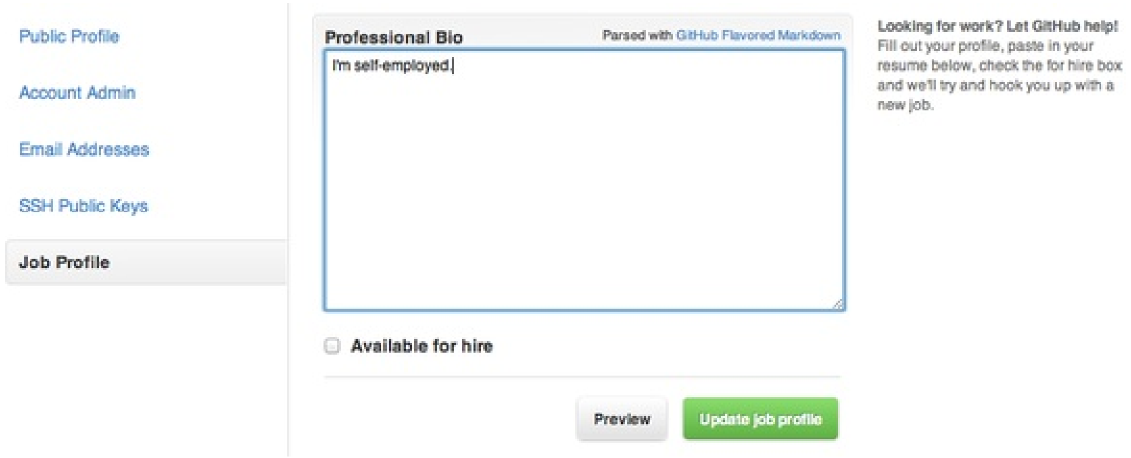


图2-9：求职信息管理

2.2. 浏览托管项目

在上一节学习了如何建立GitHub的账户，接下来在建立自己的项目托管之前，先来看看别人是怎么做的。

在GitHub中搜索字符串“GotGit”，可以搜索到我建立的一个项目，项目地址是：https://github.com/gotgit/gotgit/ 。由上至下，GotGit项目首页可以分为如下几个区域。

- 区域一：项目概要介绍及版本库URL等。

  项目GotGit托管在组织账号 gotgit 之下(项目 gotgit 最早由用户 ossxp-com 创建，现已转移到组织 gotgit 账号之下。)，并且已经有若干关注用户和派生项目。最下面一行显示版本库的访问地址，只显示了HTTP和Git-daemon两个协议的URL地址，这是因为当前用户对该版本库只具有只读权限，因此没有显示SSH协议的URL地址。



  图2-10：版本库概要信息

  使用任意一种协议均可克隆该Git版本库，但要注意只有Git 1.6.6及以上版本才支持智能HTTP协议，低版本Git则无法用HTTP协议克隆GitHub上的版本库[2]。

```
$ git clone https://github.com/gotgit/gotgit.git
```

```
或者使用Git-daemon协议。
```

```
$ git clone git://github.com/gotgit/gotgit.git
```

- 区域二：代码浏览子菜单及分支切换对话框。

  默认项目代码页（即项目首页）显示项目文件列表（即Files子菜单），如图2-11所示。右侧还显示项目gotgit/gotgit默认的分支为gh-pages而非常见的master分支。关于gh-pages分支，在“第3.5.2节 创建项目主页”会介绍该分支的神奇用途。



图2-11：代码浏览子菜单及分支

- 区域三：显示最新提交的提交说明、提交用户头像、提交时间等提交信息。右侧还显示此次提交对应的提交ID。



图2-12：提交信息

- 区域四：目录树。每个目录和文件后面还显示最后一次变更的提交说明。

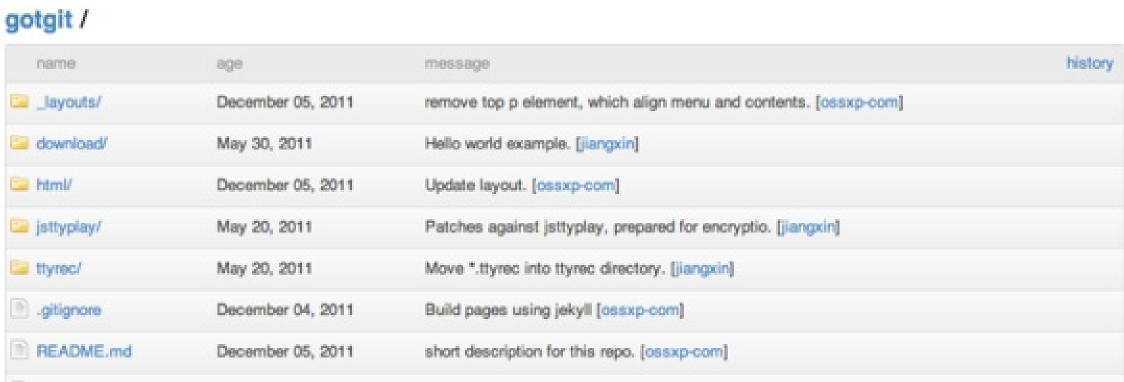


图2-13：目录树

- 区域五：根目录下的文件README.md格式化为HTML输出。

  GitHub内置了多种文本标记语言的支持，如Markdown、Textile、reStructuredText、asciidoc、Wiki等。当发现根目录下的README文件后，会根据其扩展名判断所用的标记语言类型，自动转换为HTML格式显示。



图2-14：README文件

在GitHub的页面中可以使用键盘快捷键，按下问号（?）会在弹出窗口显示当前页面可用的快捷键。

在项目的代码浏览页按下字母“w”，弹出分支切换菜单，如图2-15所示。

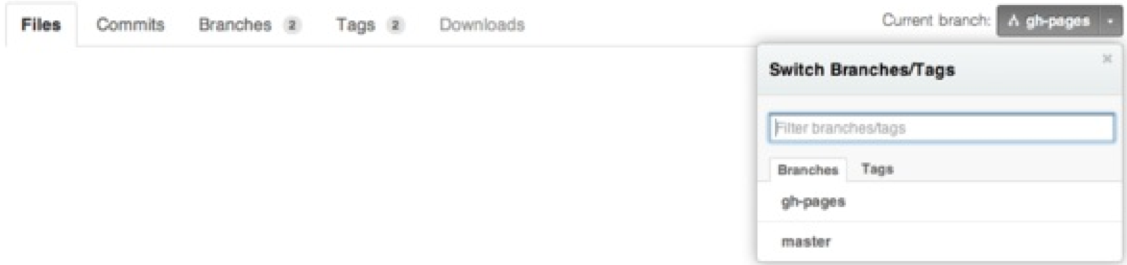


图2-15：快捷键“w”切换分支

按下字母“t”，开启目录树中文件查找和过滤。图2-16就是在按下字母“t”后，当逐一输入单词“download”时的过滤效果。

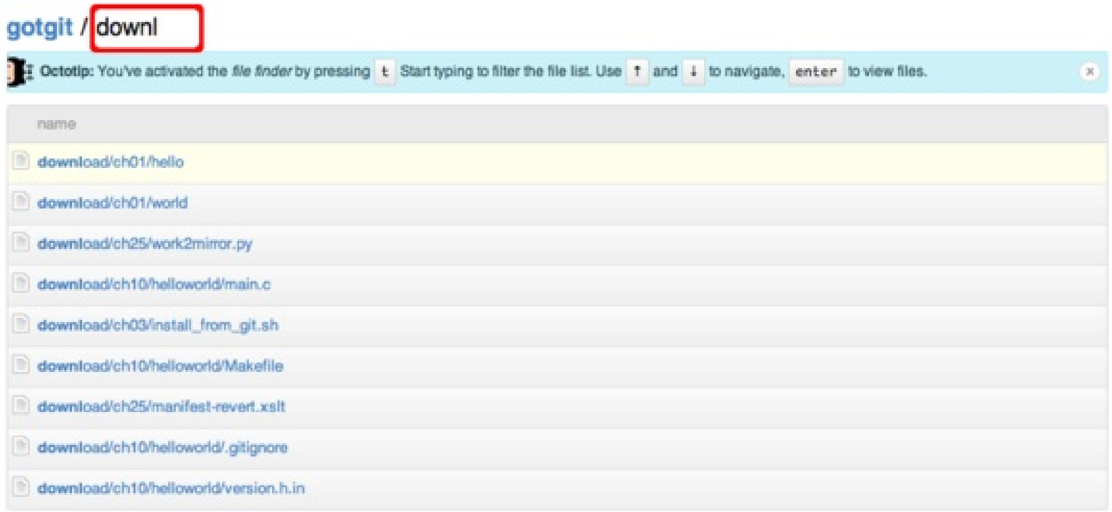


图2-16：快捷键“t”开启过滤器在目录树中搜索

点击代码浏览子菜单中的“Commits”（提交）显示版本库GotGit的提交历史，如图2-17所示。

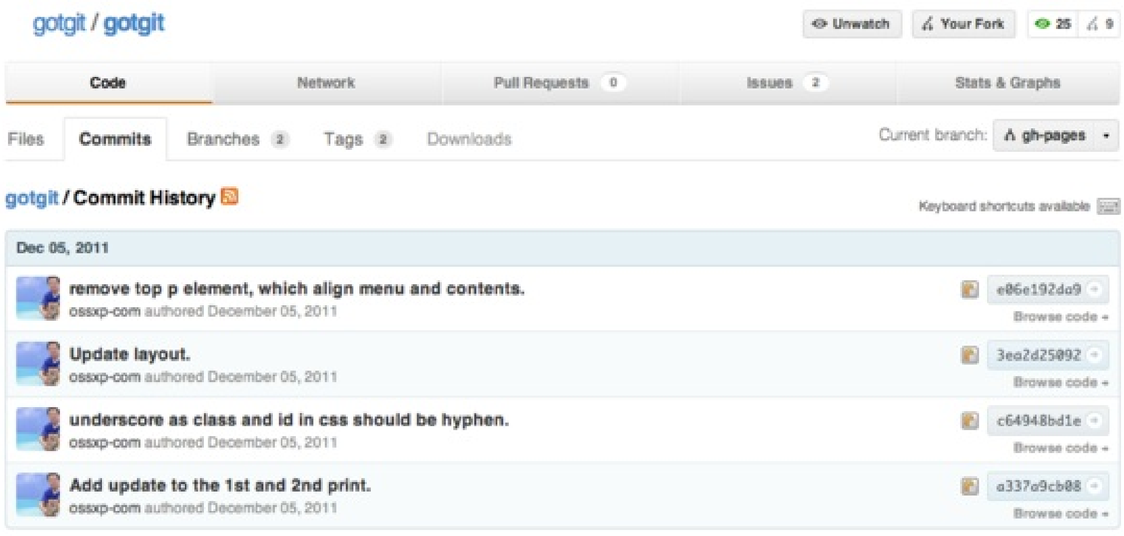


图2-17：提交历史

提交历史页面也支持快捷键，按下问号（?）或者点击页面中的键盘标志会显示快捷键帮助。其中快捷键"j"和"k"用于在提交列表中向上和向下选择提交，在选中的提交按下回车键，会显示该提交包含的文件改动差异，如图2-18所示。



图2-18：文本文件改动差异

在文本文件的差异比较中，不但将有差异的行标识出来，还将行内具体改动的字词用特殊颜色进行了标识，不由得感叹GitHub的细致入微。

GitHub还支持对图形文件的差异比较，并提供四种比较方式。在如下地址：http://git.io/image-diff (短格式URL，实际对应于: https://github.com/cameronmcefee/Image-Diff-View-Modes/commit/8e95f7)提供了一个示例提交。您可以去尝试一下不同的图形文件比较方式，以便更直观地观察图形文件前后的改动%E6%8F%90%E4%BE%9B%E4%BA%86%E4%B8%80%E4%B8%AA%E7%A4%BA%E4%BE%8B%E6%8F%90%E4%BA%A4%E3%80%82%E6%82%A8%E5%8F%AF%E4%BB%A5%E5%8E%BB%E5%B0%9D%E8%AF%95%E4%B8%80%E4%B8%8B%E4%B8%8D%E5%90%8C%E7%9A%84%E5%9B%BE%E5%BD%A2%E6%96%87%E4%BB%B6%E6%AF%94%E8%BE%83%E6%96%B9%E5%BC%8F%EF%BC%8C%E4%BB%A5%E4%BE%BF%E6%9B%B4%E7%9B%B4%E8%A7%82%E5%9C%B0%E8%A7%82%E5%AF%9F%E5%9B%BE%E5%BD%A2%E6%96%87%E4%BB%B6%E5%89%8D%E5%90%8E%E7%9A%84%E6%94%B9%E5%8A%A8)。

- 默认修改前后的两幅图片左右并排显示，如图2-19所示。

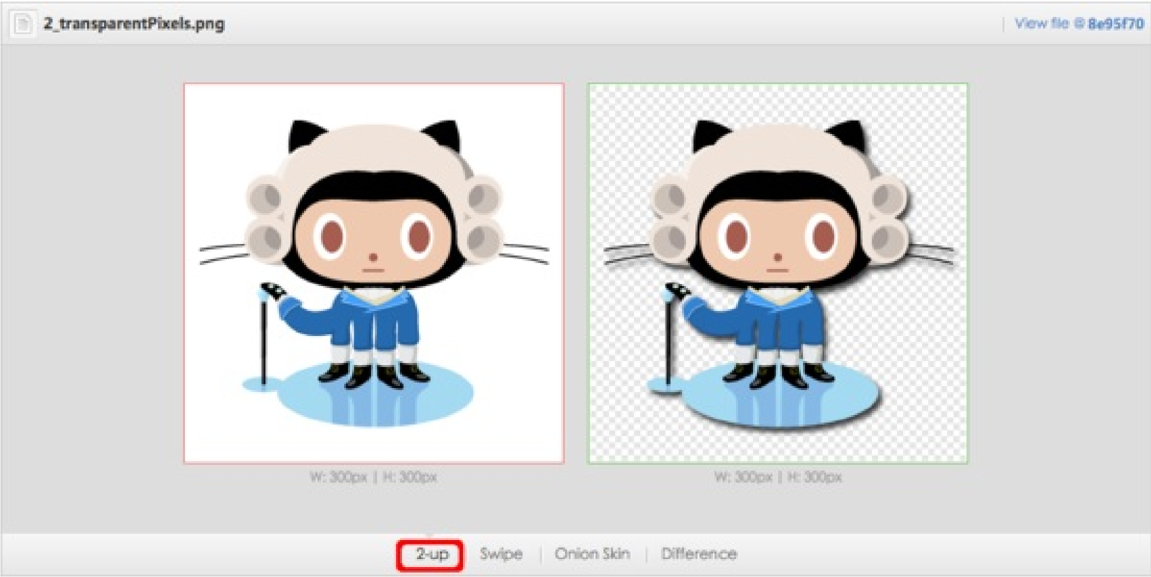


图2-19：左右并排比较图形文件差异

- 选择交换显示比较修改前后的图片，用鼠标左右拖动进度条，可以非常直观地看到图片的差异。如图2-20所示。



图2-20：交换显示图形文件比较差异

- 还提供洋葱皮和色差比较，自己动手试试吧。

网络图是GitHub的一大特色，显示一个项目的版本库被不同用户派生（Fork）后，各个版本库的派生关系。这个网络图最早使用Flash实现的，目前已经改为HTML5实现[4]。图2-21的示例网络图来自于Gitosis项目[5]。

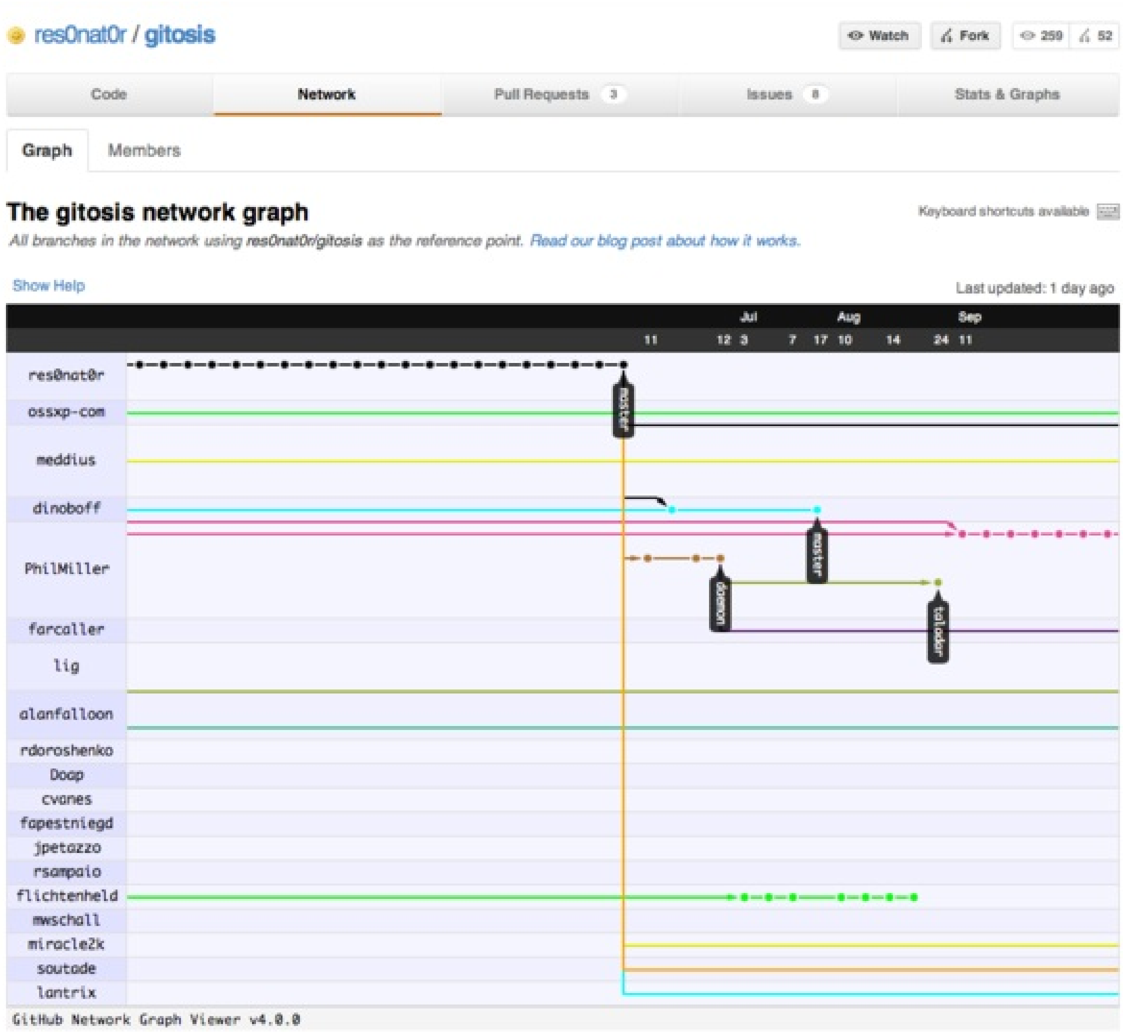


图2-21：Gitosis项目网络图

Pull Requests（拉拽请求）是派生（Fork）版本库的开发者向项目贡献提交的方法。如图2-22所示，GotGit项目目前没有未被处理的Pull Request，但是可以看到有一个已经关闭的Pull Request请求。

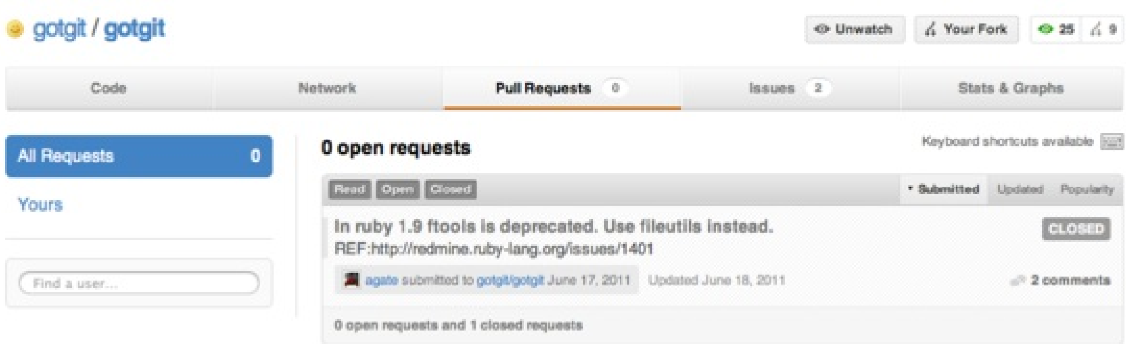


图2-22：Pull Requests界面

这个Pull Request是GitHub用户agate发现了GotGit脚本中一个和ruby1.9不兼容的Bug，当我把agate派生版本库中的提交合并到GotGit版本库后，该Pull Request自动关闭。整个Pull Request的变更记录如图2-23所示。

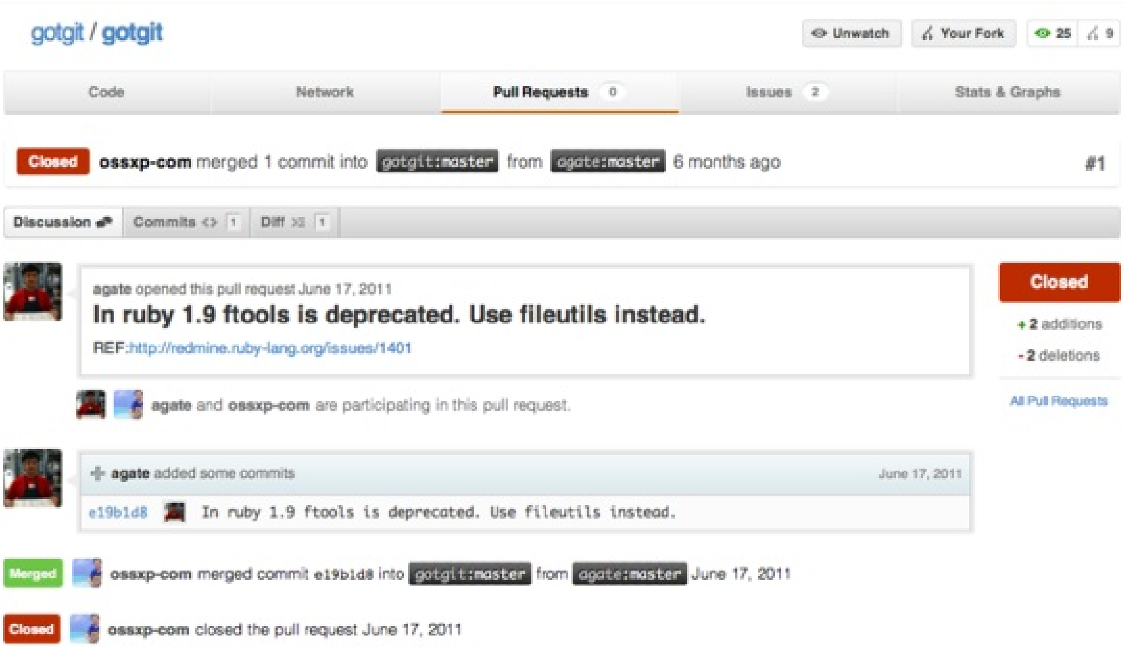


图2-23：Pull Request的变更历史

缺陷追踪（Issue）也是GitHub工作流中一个重要的组件。GotGit项目用缺陷跟踪系统帮助维护《Git权威指南》一书的勘误，图2-24可以看到当前有2个打开的问题和9个已关闭的问题。

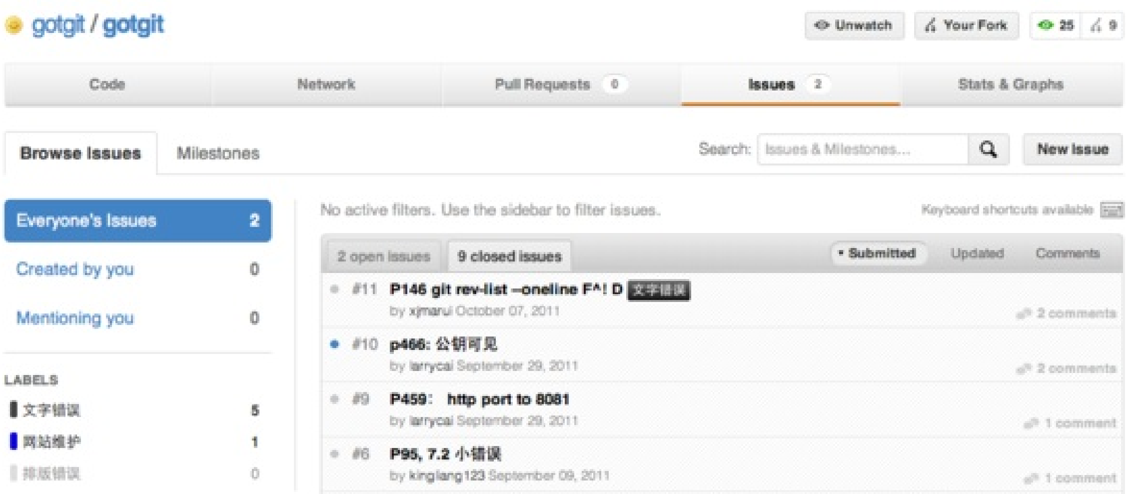


图2-24：缺陷追踪

GitHub还为项目提供报表分析。图2-25是GotGit项目中用到的开发语言分布图。

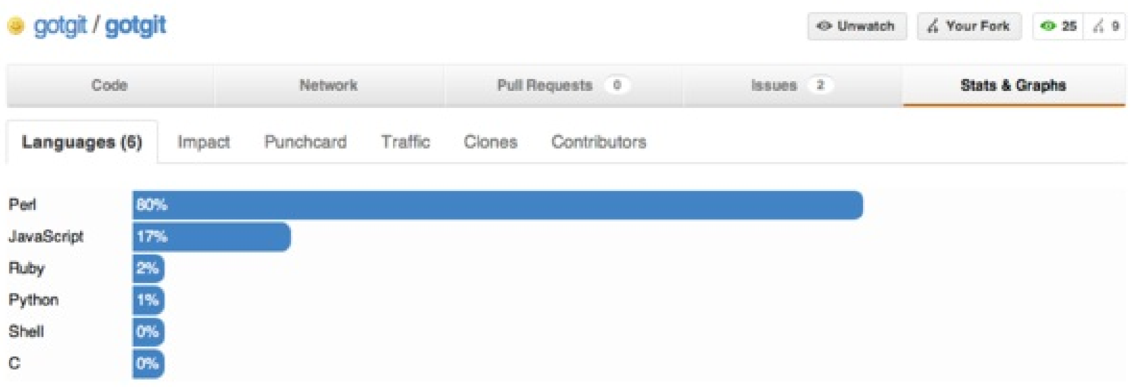


图2-25：GotGit项目开发语言分布图

图2-26是开发者对GotGit项目贡献分布图。

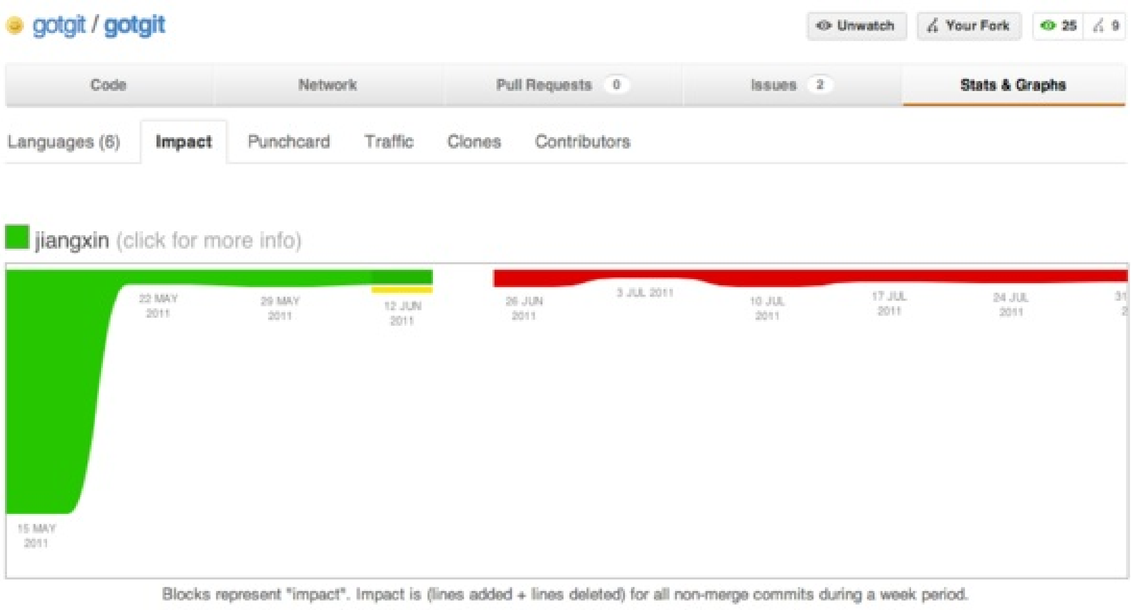


图2-26：GotGit项目贡献分布图

2.3. 社交网络

社交网络的一大特征就是用户间的相互关注，从而形成朋友圈或媒体圈，实现便捷的信息分享和传播。GitHub支持项目级别及用户级别的关注。

关注一个项目很简单，只需点击项目名称右侧的“Watch”按钮。



图2-27：项目的关注按钮

添加对项目的关注后，点击页面左上角的“github”文字图标进入仪表板（Dashboard）页面，如图2-28所示。

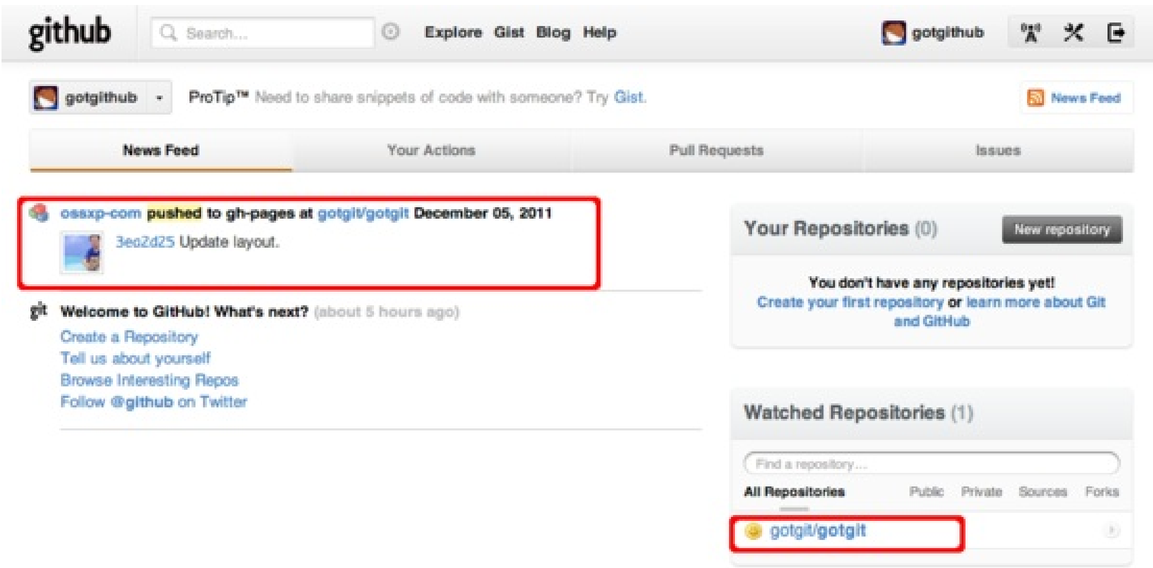


图2-28：关注项目在仪表板页的显示

仪表板页面的左侧显示所关注项目的最新动态，右侧会列表显示关注的项目列表。

GitHub还可以关注用户。访问 https://github.com/mojombo 可以看到mojombo（GitHub创始者之一）的用户页，关注他只需点击图2-29中的“Follow”按钮。从mojombo的用户页还可以看到majombo关注的开发者，可以以此扩大GitHub朋友圈。

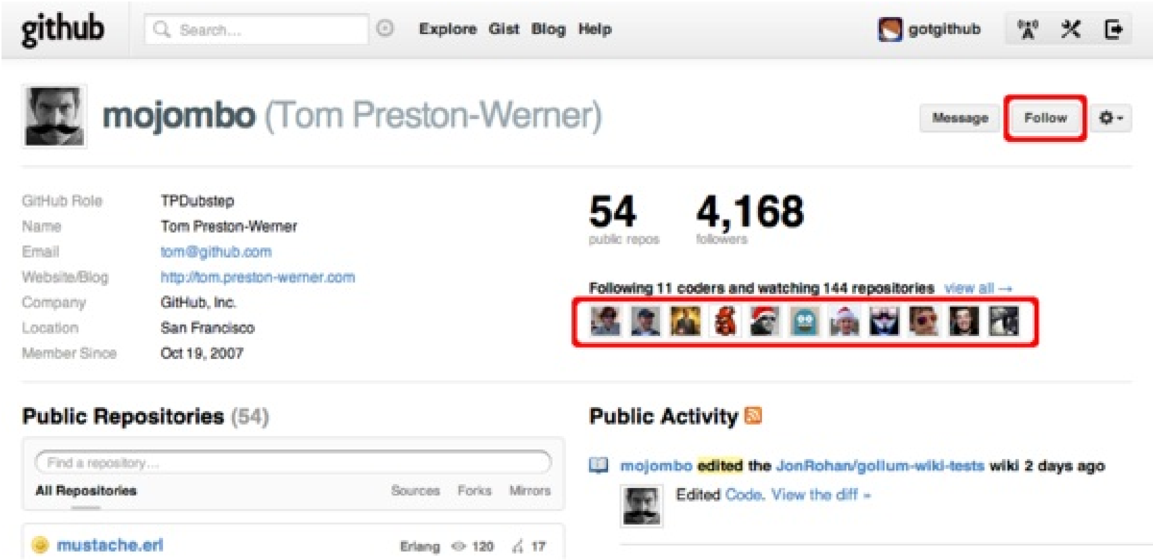


图2-29：mojombo的用户页

GitHub仪表板页面，有一个“RSS Feed”链接，如图2-30所示。点击该链接可以使用RSS客户端（如Google Reader）订阅，实现无需登录GitHub即可访问所关注的项目和人的动态。



图2-30：RSS Feed

RSS中可能包括隐私信息，如私有版本库的更新信息，那么RSS订阅是如何保护个人隐私呢？难道需要口令认证么？查看一下RSS订阅的URL，你会看到类似如下格式的URL地址：

```
https://github.com/gotgithub.private.atom?token=681a8a5a38419ecfb80f8633d4cb4e16
```

原来RSS订阅用到了API Token进行身份认证，即保障了个人RSS的私密性，又避免直接使用明文口令导致的密码泄露。关于API Token，参见本章第2.1节中相关介绍。

3. 项目托管

本章介绍如何在GitHub上创建一个新项目，包括创建版本库及为项目设计主页等。

3.1. 创建新项目

3.1.1. 新版本库即是新项目

在GitHub，一个项目对应唯一的Git版本库，创建一个新的版本库就是创建一个新的项目。访问仪表板（Dashboard）页面，如图3-1，可以看到关注的版本库中已经有一个，但自己的版本库为零。在显示为零的版本库列表面板中有一个按钮“New Repository”，点击该按钮开始创建新版本库。

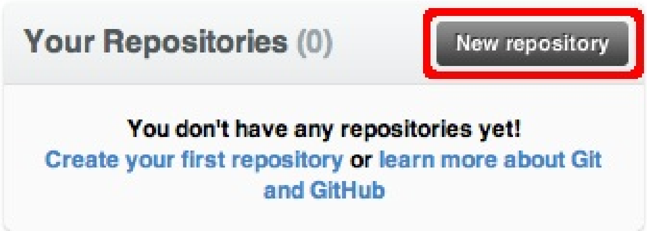


图3-1：版本库列表面板

新建版本库的界面如图3-2所示。

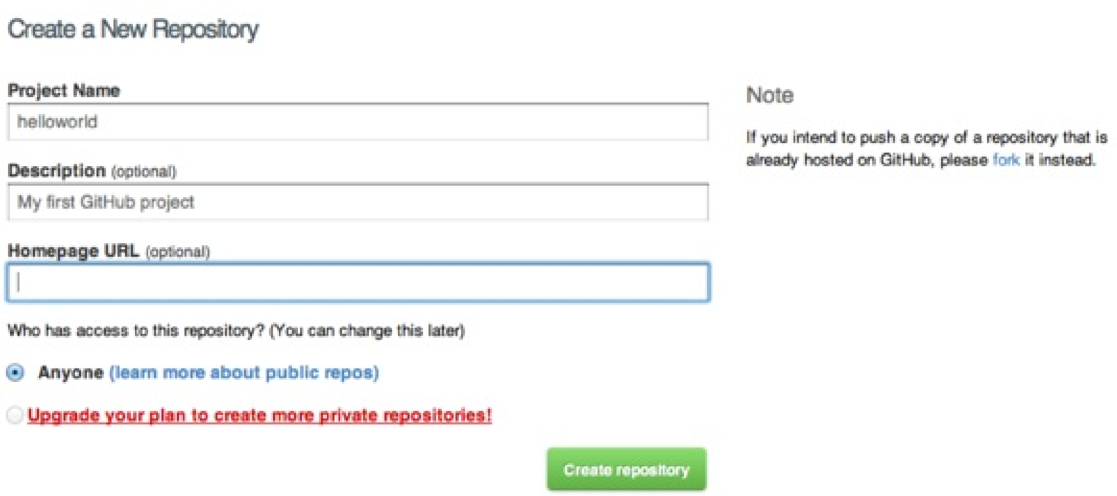


图3-2：创建新项目

我们为新建立的版本库命名为“helloworld”，相应的项目名亦为“helloworld”，创建完毕后访问项目页，提示版本库尚未初始化，并给出如何初始化版本库的帮助，如图3-3所示。

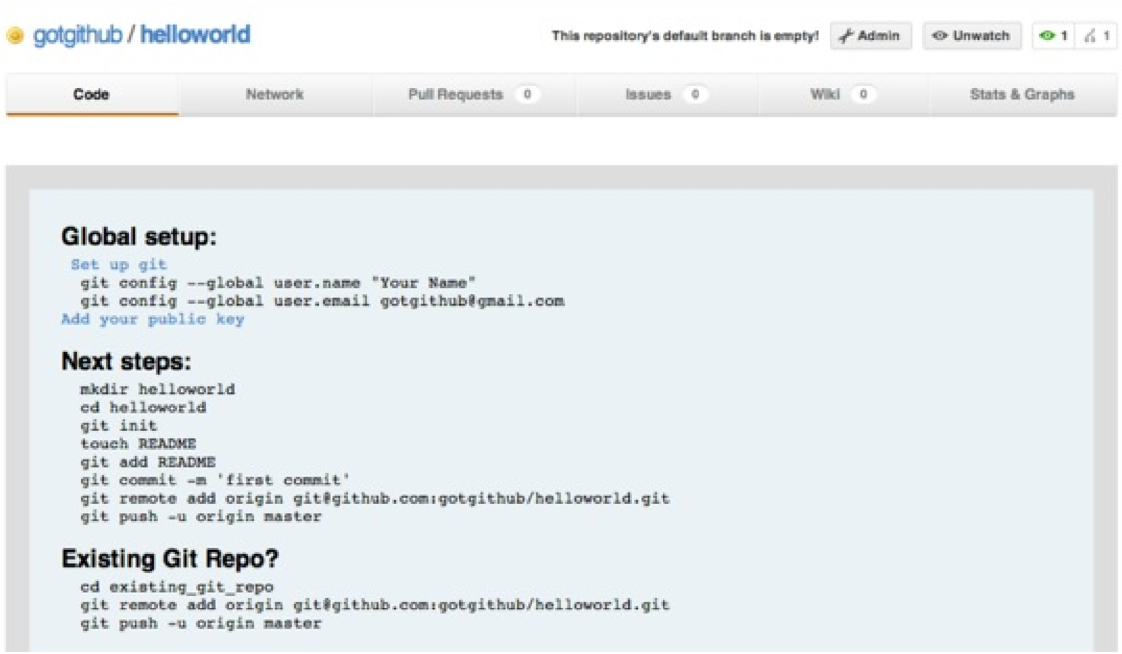


图3-3：项目尚未初始化

在图3-3中可以看到访问协议增加了一个支持读写的SSH协议，访问地址为：git@github.com:gotgithub/helloworld.git。注意任何GitHub用户均可使用该URL访问此公开版本库，但只有版本库建立者gotgithub具有读写权限，其他人只有只读权限。在初始化版本库之前，最好先确认是否是用正确的公钥进行认证，如下：

```
$ ssh -T git@github.com
Hi gotgithub! You've successfully authenticated, but GitHub does not provide shell access
```

3.1.2. 版本库初始化[]

如果是从头创建版本库，可以采用先克隆，建立提交数据，最后再通过推送完成GitHub版本库的初始化。步骤如下：

- 克隆版本库。

  克隆过程会显示警告，不过这个警告可以忽略，因为GitHub创建的版本库本来就是一个空白的版本库。

```
$ git clone git@github.com:gotgithub/helloworld.git
Cloning into 'helloworld'...
warning: You appear to have cloned an empty repository.
```

- 创建文件README.md（注：以扩展名.md，.mkd，.mkdn，.mdown，.markdown等为结尾的文件，均以Markdown标记语言语法进行解析并显示。）。

下面是一段示例文字，把这段文字保存为文件README.md，该文件的内容将会直接显示在项目首页中（显示效果参见后面的图3-5）。

```
# 我的第一个GitHub项目

这是项目 [helloworld](https://github.com/gotgithub/helloworld) ，
欢迎访问。

这个项目的版本库是 **Git格式** ，在 Windows、Linux、Mac OS X
平台都有客户端工具可以访问。虽然版本库只提供Git一种格式，
但是你还是可以用其他用其他工具访问，如 ``svn`` 和 ``hg`` 。

## 版本库地址

支持三种访问协议：

* HTTP协议: `https://github.com/gotgithub/helloworld.git` 。
* Git协议: `git://github.com/gotgithub/helloworld.git` 。
* SSH协议: `ssh://git@github.com/gotgithub/helloworld.git` 。

## 克隆版本库

操作示例：

$ git clone git://github.com/gotgithub/helloworld.git
```

上面这段文字采用Markdown格式，您也可以使用其他支持的格式，只要确保README文件使用正确的扩展名。本书附录部分介绍了Markdown及其他GitHub支持的标记语言。关于Markdown，目前我们只需知道这一个易于识别和理解的纯文本格式，可以方便的转换为HTML。Markdown语法非常像我们在写邮件（纯文本）时用空行来分隔段落、用星号开启列表、用缩进表示引用内容等等。

- 添加README.md文件并提交。

```
$ git add README.md
$ git commit -m "README for this project."
```

- 向GitHub推送，完成版本库初始化。

```
$ git push origin master
```

然后查看GitHub上新建项目的首页。项目首页的上半部分可见版本库包含了一个新的提交，以及版本库目录树中包含的文件，如图3-4所示。

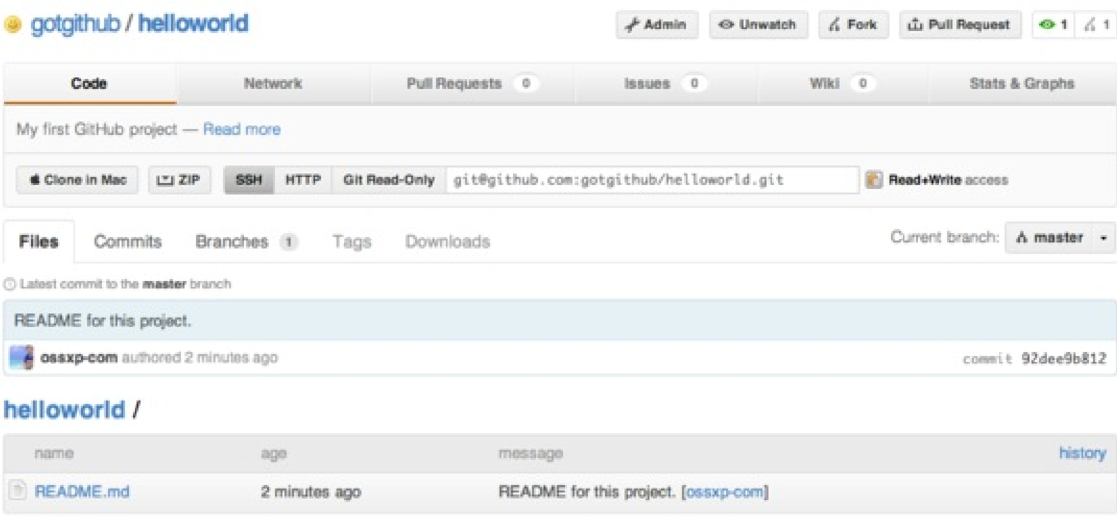


图3-4：完成推送后的项目首页上半部分

在项目首页的下半部分，会看到README.md文件被转换为HTML显示，如图3-5所示。


README.md

我的第一个GitHub项目

这是项目 helloworld ，欢迎访问。

这个项目的版本库是 Git格式 ，在 Windows、Linux、Mac OS X 平台都有客户端工具可以访问。虽然版本库只提供Git一种格式，但是你还是可以用其他用其他工具访问，如 svn 和 hg 。

版本库地址

支持三种访问协议：

- HTTP协议：https://github.com/gotgithub/helloworld.git 。
- Git协议：git://github.com/gotgithub/helloworld.git 。
- SSH协议：ssh://git@github.com/gotgithub/helloworld.git 。

克隆版本库

操作示例：

```
$ git clone git://github.com/gotgithub/helloworld.git
```

图3-5：完成推送后的项目首页下半部分

3.1.3. 从已有版本库创建

如果在GitHub项目初始化之前，数据已经存在于本地版本库中，显然像上面那样先克隆、再提交、后推送的方法就不适宜了。应该采用下面的方法。

为试验新的版本库初始化方法，先把刚刚新建的测试项目“helloworld”删除，同时也将本地工作区中克隆的“helloworld”删除。警告：删除项目的操作非常危险，不可恢复，慎用。

- 点击项目首页中项目名称旁边的“Admin”按钮进入项目管理页，再点击页面最下方的删除版本按钮，如图3-6所示。

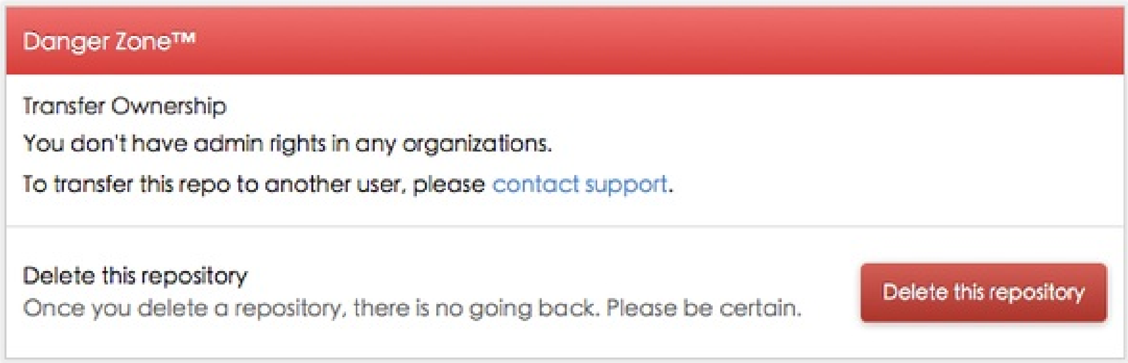


图3-6：删除项目

* 然后再重建版本库“helloworld”，如本章一开始图3-2所示。

接下来使用下面的步骤完成“helloworld”版本库的初始化。

* 本地建立一个Git版本库。

```
$ mkdir helloworld
$ cd helloworld
$ git init
```

* 然后在版本库中添加示例文件，如README.md文件，内容同前。

```
$ git add README.md
$ git commit -m "README for this project."
```

* 为版本库添加名为origin的远程版本库。

```
$ git remote add origin git@github.com:gotgithub/helloworld.git
```

* 执行推送命令，完成GitHub版本库的初始化。注意命令行中的-u参数，在推送成功后自动建立本地分支与远程版本库分支的追踪。

```
$ git push -u origin master
```

3.2. 操作版本库

3.2.1. 强制推送

细心的读者可能从图3-4已经看出，显示的提交者并非gotgithub用户，而是一个名为ossxp-com的用户，这是因为GitHub是通过提交中的邮件地址来对应到GitHub用户的。看看提交说明：

```
$ git log --pretty=fuller
commit 92dee9b8125afc9a606394ed463f9f264f2d3d58
Author:     Jiang Xin
AuthorDate: Wed Dec 14 14:52:40 2011 +0800
Commit:     Jiang Xin
CommitDate: Wed Dec 14 14:52:40 2011 +0800

    README for this project.
```

原来提交用户设置的邮件地址并非gotgithub用户设置的邮件地址。补救办法就是对此提交进行修改，然后强制推送到GitHub。

- 重新设置user.name和user.email配置变量。

  因为gotgithub是一个仅在本书使用的示例账号，我可不想影响本地其他项目的提交，因此下面的设置命令没有使用--global参数，只对本地helloworld版本库进行设置。

```
$ git config user.name "Jiang Xin"
$ git config user.email "gotgithub@gmail.com"
```

- 执行Git修补提交命令。

  注意使用参数--reset-author会将提交信息中的属性Author连同AuthorDate一并修改，否则只修改Commit和CommitDate。参数-C HEAD维持提交说明不变。

```
$ git commit --amend --reset-author -C HEAD
```

- 查看提交日志，发现提交者信息和作者信息都已经更改。

```
$ git log --pretty=fuller
commit e1e52d99fa71fd6f606903efa9da04fd0055fca9
Author:     Jiang Xin
AuthorDate: Wed Dec 14 15:05:47 2011 +0800
Commit:     Jiang Xin
CommitDate: Wed Dec 14 15:05:47 2011 +0800

    README for this project.
```

- 直接推送会报错。

  错误信息中出现non-fast-forword（非快进式推送），含义为要推送的提交并非继远程版本库最新提交之后的提交，推送会造成覆盖导致服务器端有数据（提交）会丢失。

```
$ git push
To git@github.com:gotgithub/helloworld.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:gotgithub/helloworld.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

- 使用强制推送。

  对于此例，考虑到还没有其他人关注helloworld这个刚刚建立的示例项目，显然不需要向上面命令的错误信息所提示的那样先执行git pull合并上游版本库再推送，而是选择强制推送，以新的修补提交覆盖包含错误提交者ID的提交。

```
$ git push -f
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 629 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:gotgithub/helloworld.git
 + 92dee9b...e1e52d9 master -> master (forced update)
```

完成强制推送后，再查看GitHub项目页面，会发现提交者已经显示为gotgithub用户。如图3-7所示。

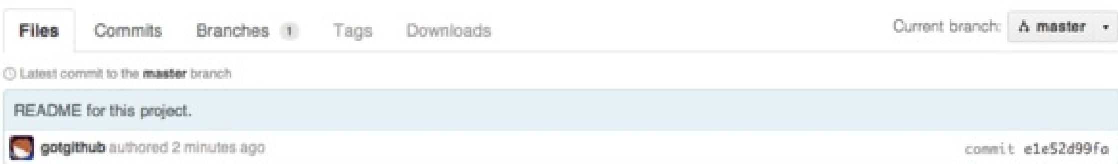


图3-7：强制更新后，提交者已更改

3.2.2. 新建分支

Git的分支就是保存在 `.git/refs/heads/` 命名空间下的引用。引用文件的内容是该分支对应的提交ID。当前版本库中的默认分支master就对应于文件 `.git/refs/heads/master` 。

若在GitHub版本库中创建分支，首先要在本地版本库中创建新的分支（即引用），然后用推送命令将本地创建的新的引用连同所指向的提交推送到GitHub版本库中完成GitHub上分支的创建。操作如下：

- 本地版本库中建立新分支mybranch1。

创建分支有多种方法，如使用git branch命令，但最为便捷的就是git checkout-b命令，同时完成新分支的创建和分支切换。

```
$ git checkout -b mybranch1
Switched to a new branch 'mybranch1'
```

- 为了易于识别，添加一个新文件hello1，并提交。

```
$ touch hello1
$ git add hello1
$ git commit -m "add hello1 for mark."
[mybranch1 f46a284] add hello1 for mark.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello1
```

- 通过推送命令，将本地分支mybranch1推送到GitHub远程版本库，完成在GitHub上的新分支创建。

```
$ git push -u origin mybranch1
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 281 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:gotgithub/helloworld.git
 * [new branch]      mybranch1 -> mybranch1
Branch mybranch1 set up to track remote branch mybranch1 from origin.
```

在GitHub上查看版本库，会看到新增了一个分支mybranch1，不过默认分支仍为master，如图3-8所示。

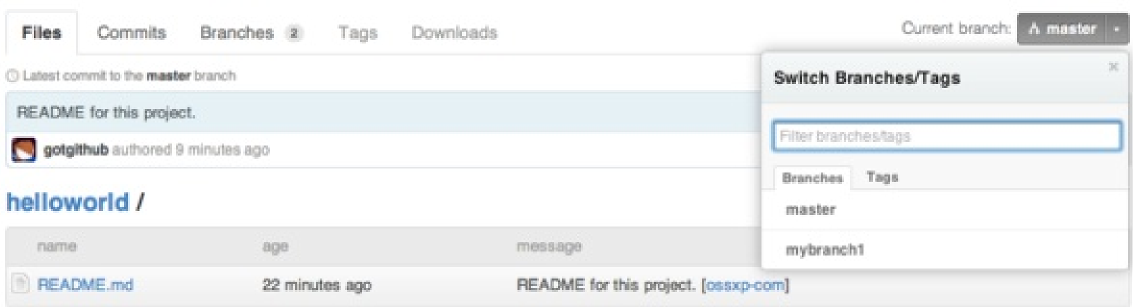


图3-8：版本库新增了一个分支

3.2.3. 设置默认分支

可以改变GitHub上版本库显示的默认分支，如果版本库包含多个分支的话。例如修改版本库的默认分支为mybranch1，点击项目名称旁边的“Admin”按钮，修改项目的默认分支。如图3-9所示。

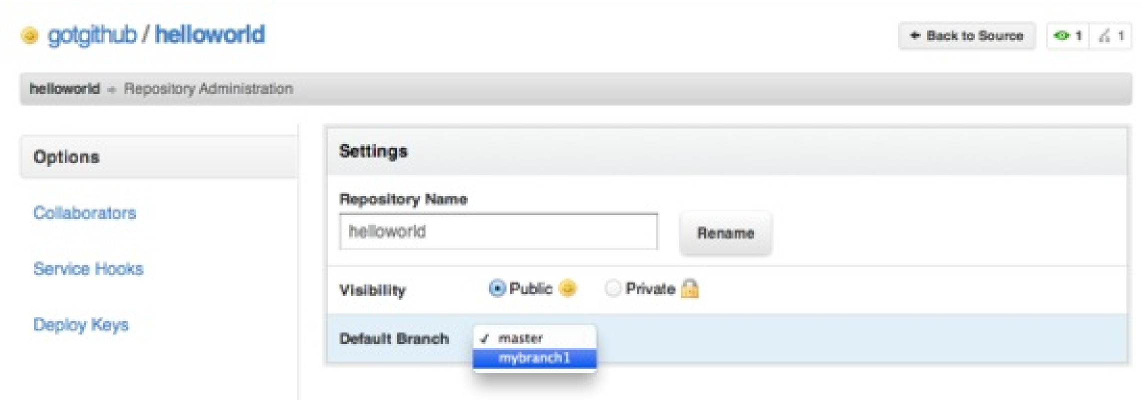


图3-9：设置缺省分支

设置了GitHub默认分支后，如果再从GitHub克隆版本库，本地克隆后版本库的默认分支也将改变。

```
$ git clone git@github.com:gotgithub/helloworld.git helloworld-nb
Cloning into 'helloworld-nb'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 6 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd helloworld-nb
$ git branch
* mybranch1
```

实际上修改GitHub上版本库的默认分支，就是将GitHub版本库的头指针HEAD指向了其他分支，如mybranch1分支。这可以从下面命令看出。

```
$ git branch -r
  origin/HEAD -> origin/mybranch1
  origin/master
  origin/mybranch1
```

也可以从git ls-remote命令看出头指针HEAD和引用refs/heads/mybranch1指向同一个对象的哈希值。

```
$ git ls-remote
From git@github.com:gotgithub/helloworld.git
f46a28484adb6c1b4830eb4df582325c740e9d6c        HEAD
e1e52d99fa71fd6f606903efa9da04fd0055fca9        refs/heads/master
f46a28484adb6c1b4830eb4df582325c740e9d6c        refs/heads/mybranch1
```

3.2.4. 删除分支

删除当前工作分支会报错。例如下面的命令试图分支mybranch1，但没有成功：

```
$ git branch -d mybranch1
error: Cannot delete the branch 'mybranch1' which you are currently on.
```

错误信息显示不能删除当前工作分支。因此先切换到其他分支，例如从GitHub版本库中取出master分支并切换。

```
$ git checkout master
```

可以看出新的工作分支为master分支。

```
$ git branch
* master
  mybranch1
```

现在可以删除mybanch1分支。下面的命令之所以使用-D参数，而非-d参数，是因为Git在删除分支时为避免数据丢失，默认禁止删除尚未合并的分支。参数-D则可强制删除尚未合并的分支。

```
$ git branch -D mybranch1
Deleted branch mybranch1 (was f46a284).
```

现在只是本地分支被删除了，远程GitHub服务器上的mybranch1分支尚在。删除远程GitHub版本库中的分支就不能使用git branch命令，而是要使用git push命令，不过在使用推送分支命令时要使用一个特殊的引用表达式（冒号前为空）。如下：

```
$ git push origin :mybranch1
remote: error: refusing to delete the current branch: refs/heads/mybranch1
To git@github.com:gotgithub/helloworld.git
 ! [remote rejected] mybranch1 (deletion of the current branch prohibited)
error: failed to push some refs to 'git@github.com:gotgithub/helloworld.git'
```

为什么删除远程分支出错了呢？是因为没有使用强制推送么？

实际上即使使用强制推送也会遇到上面的错误。GitHub发现要删除的mybranch1分支是远程版本库的缺省分支，因而禁止删除。重新访问GitHub的项目管理页面，将缺省分支设置回master分支，参照图3-9。然后再次执行如下命令，即可成功删除分支。

```
$ git push origin :mybranch1
To git@github.com:gotgithub/helloworld.git
 - [deleted]         mybranch1
```

执行git ls-remote命令可以看到GitHub远程版本库已经不存在分支mybranch1。

```
$ git ls-remote git@github.com:gotgithub/helloworld.git
From git@github.com:gotgithub/helloworld.git
e1e52d99fa71fd6f606903efa9da04fd0055fca9        HEAD
e1e52d99fa71fd6f606903efa9da04fd0055fca9        refs/heads/master
```

3.2.5. 里程碑管理

里程碑即tag，其管理和分支管理非常类似。里程碑和分支一样也是以引用的形式存在的，保存在.git/refs/tags/路径下。引用可能指向一个提交，但也可能是其他类型（Tag对象）。

- 轻量级里程碑：用git tag [] 命令创建，引用直接指向一个提交对象。
- 带说明的里程碑：用git tag -a [] 命令创建，并且在创建时需要提供创建里程碑的说明。Git会创建一个tag对象保存里程碑说明、里程碑的指向、创建里程碑的用户等信息，里程碑引用指向该Tag对象。
- 带签名的里程碑：用git tag -s [] 命令创建。是在带说明的里程碑的基础上引入了PGP签名，保证了所创建的里程碑的完整性和不可拒绝性。

下面演示一下里程碑的创建和管理。

- 先在本地创建一个新提交。

```
$ touch hello1
$ git add hello1
$ git commit -m "add hello1 for mark."
```

- 本地创建里程碑mytag1、mytag2和mytag3。

```
$ git tag -m "Tag on initial commit" mytag1 HEAD^
$ git tag -m "Tag on new commit"     mytag2
$ git tag mytag3
```

- 查看新建立的里程碑。

```
$ git tag -l -n1
mytag1          Tag on initial commit
mytag2          Tag on new commit
mytag3          add hello1 for mark.
```

- 将本地里程碑推送到GitHub远程版本库。

```
$ git push origin refs/tags/*
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 548 bytes, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@github.com:gotgithub/helloworld.git
 * [new tag]         mytag1 -> mytag1
 * [new tag]         mytag2 -> mytag2
 * [new tag]         mytag3 -> mytag3
```

- 删除本地里程碑。

```
$ git tag -d mytag3
Deleted tag 'mytag3' (was c71231c)
```

- 删除GitHub远程版本库中的里程碑。

```
$ git push origin :mytag3
To git@github.com:gotgithub/helloworld.git
  [deleted]         mytag3
```

此时查看GitHub上的项目页，会看到已有两个里程碑，如图3-10所示。

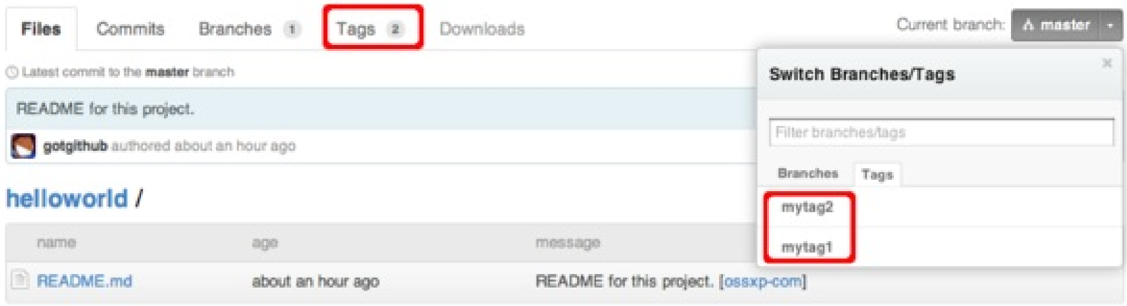


图3-10：里程碑列表

3.3. 公钥认证管理

开发者向GitHub版本库写入最常用到的协议是SSH协议，因为SSH协议使用公钥认证，可以实现无口令访问，而若使用HTTPS协议每次身份认证时都需要提供口令(可以通过在文件 `~/.netrc` 中写入明文口令实现使用 HTTPS 协议时也能自动完成认证。具体格式参见ftp命令的MAN手册中相关介绍)。使用SSH公钥认证，就涉及到公钥的管理。

3.3.1. 用户级公钥管理

开发者可能会从不止一台电脑访问GitHub中的版本库（用SSH协议），因不同的电脑有不同的公钥/私钥对，这就需要为GitHub账号添加多个公钥。点击账号设置中的“SSH Public Keys”进入SSH公钥管理界面，如图3-11所示。

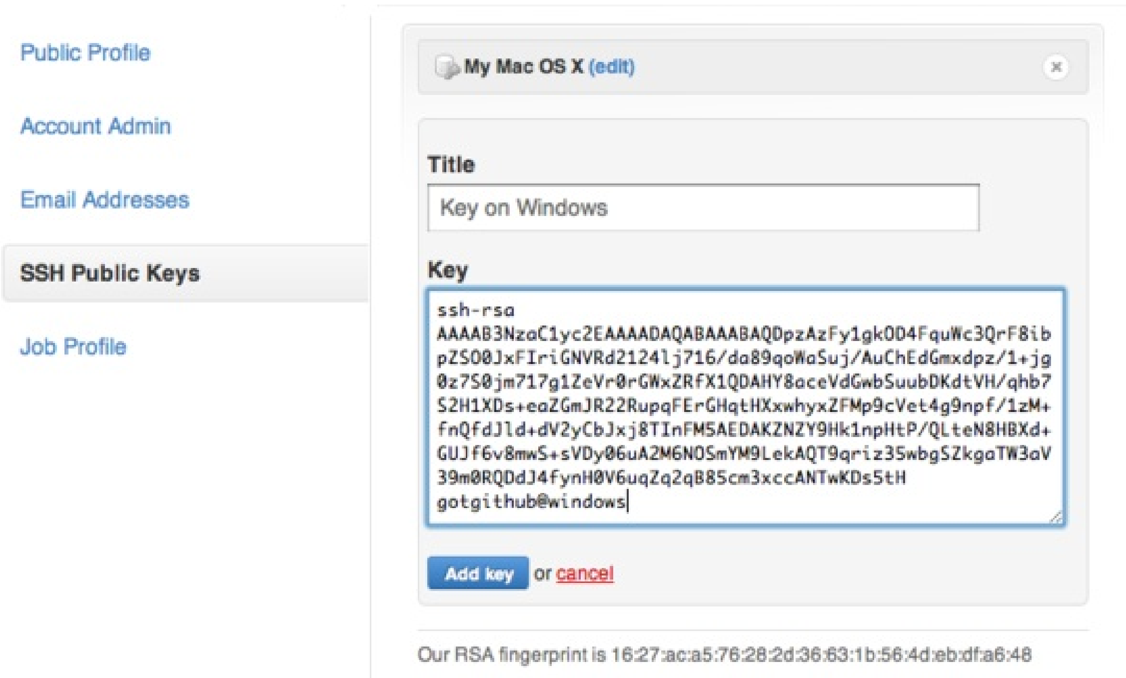


图3-11：SSH多公钥管理

如图3-11，在创建gotgithub账号一开始，就手工添加了名为“My Mac OS X”的公钥，显然这是为苹果电脑准备的。图中正在添加的名为“Key on Windows”是为Windows环境下使用SSH协议访问GitHub准备的公钥。

当添加了新的公钥后，无论是从哪台电脑（苹果或PC）用SSH协议访问版本库时都拥有相同授权，即都是以gotgithub账号身份来访问。例如用户级公钥访问GitHub的SSH服务，在提示信息中会显示用户ID，如下：

```
$ ssh -T git@github.com
Hi gotgithub! You've successfully authenticated, but GitHub does not provide shell access
```

3.3.2. 项目级公钥管理

多增加一个用户级别的公钥，就意味着可以从另外一台电脑访问该用户所有版本库。但有时只希望从某台电脑上向某一个版本库“写入”，其他版本库则不可写，这可以通过设置版本库级别的公钥认证实现。

以项目管理者（创建者）身份登录GitHub，例如以gotgithub用户身份访问gotgithub/helloworld版本库，进入到项目的管理页面，选择菜单中的“Deploy Keys”，即可设置项目级别公钥。如图3-12所示。

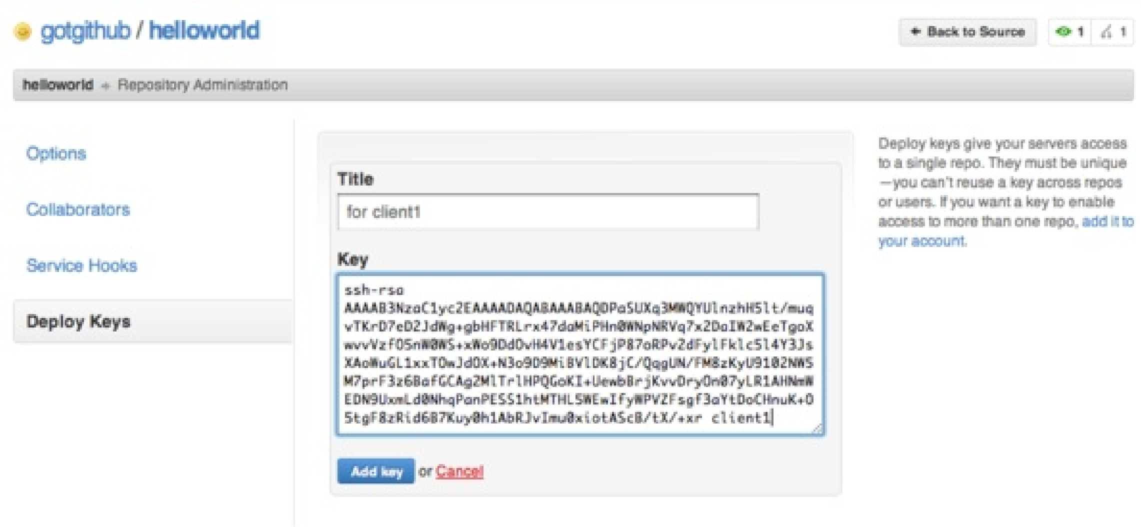


图3-12：项目级公钥管理

就像一个用户可以设置多个用户级公钥一样，也可以为一个项目设置多个项目级公钥。无论是项目级公钥还是用户级公钥都有同样的限制：一个公钥只能使用一次。

当使用项目级公钥访问GitHub的SSH服务，会在提示信息中显示版本库ID而非用户ID。如下的命令输出中显示了版本库IDgotgithub/helloworld。

```
$ ssh -i ~/.ssh/deploy-key -T git@github.com
Hi gotgithub/helloworld! You've successfully authenticated, but GitHub does not provide s
```

3.4. 版本库钩子扩展

通过钩子扩展，GitHub托管的版本库可以和外部应用实现整合。整合的接口完全开放，开发者可以访问GitHub的开源项目 github/github-services 开发新的应用整合脚本。目前GitHub已经支持超过50个外部应用的整合，在这里恕不一一列举，仅以helloworld项目为例，介绍几个常见应用的整合。

3.4.1. 邮件通知功能

配置邮件通知，可以实现新提交推送至版本库时，发送通知邮件。在版本库的管理界面，选择“Service Hooks”中的Email进入邮件通知配置界面，如图3-15所示。配置界面很简单，写上邮件地址，选择激活即可。为了便于整个团队都能收到通知邮件，可以将收件地址设置为一个邮件列表。如果选择“Send From Author”，邮件的发件者显示为提交者的邮件地址，否则发件者为noreply@github.com。

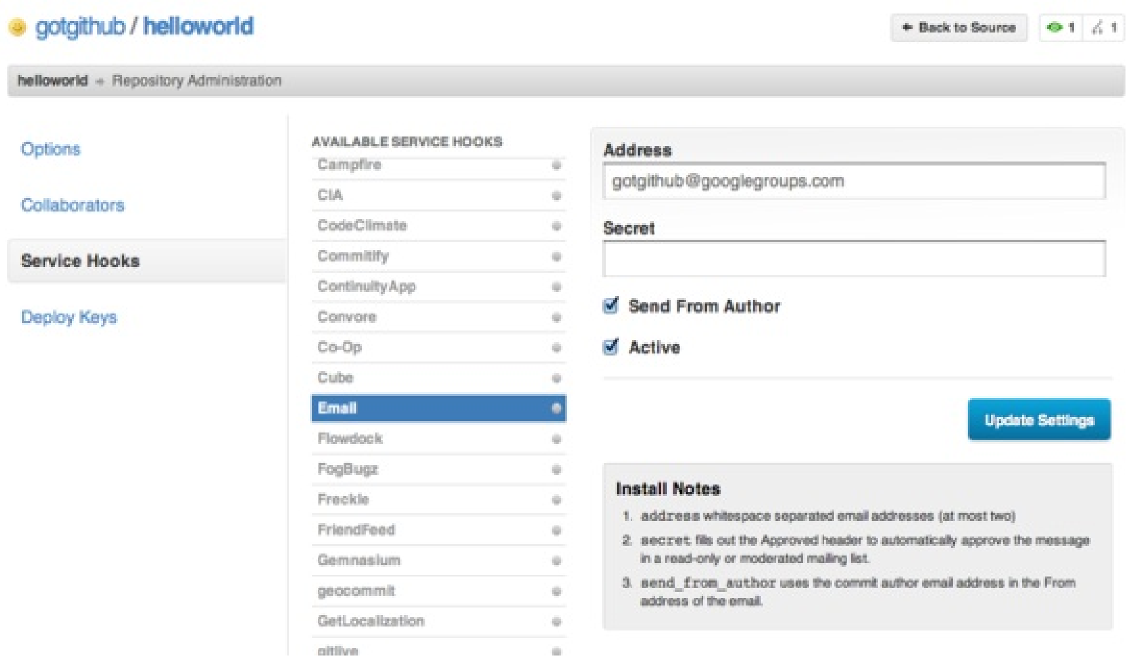


图3-15：邮件通知功能配置

邮件通知配置生效后，当有新提交推送到版本库时，会发出通知邮件，如图3-16所示。



图3-16：提交触发邮件通知

3.4.2. 和Redmine整合

Redmine是一个开源的项目管理平台，用于项目的需求管理和缺陷跟踪。Redmine可以和多种版本库（包括Git）整合，可以直接通过Web界面浏览Git提交，还实现了提交和问题的关联。

Redmine需要周期性地扫描版本库，以便更新内置数据库及建立提交和问题的关联。通常是以计划任务（crontab）的方式实现版本库的周期性扫描，这导致Redmine中版本库更新会存在一定的延迟。GitHub提供的Redmine整合的钩子脚本能够在GitHub版本库更新后，通过WebService触发Redmine主动扫描Git版本库获取更新。

GitHub提供的Redmine整合的配置界面如图3-17所示。

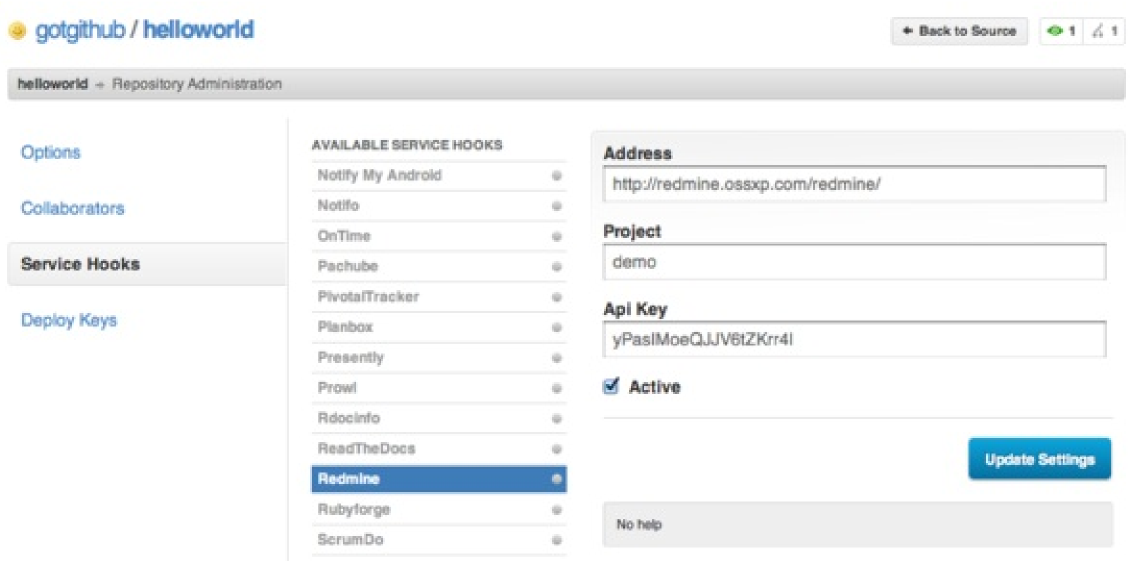


图3-17：与Redmine整合

图3-17中的地址是Redmine部署的URL地址，项目ID是Redmine中的相关项目（如果不填写则更新所有项目），而“Api Key”并非GitHub项目中配置的Api Key，而是Redmine中为版本库更新配置的全局Api Key。相应的Redmine配置界面如图3-18所示。



图3-18：Redmine中的API Key配置

3.5. 建立主页

很多开源项目托管平台都支持为托管的项目建立主页，但主页的维护方式都没有GitHub这么酷。大多数托管平台无非是开放一个FTP或类似服务，用户把制作好的网页或脚本上传了事，而在GitHub用户通过创建特殊名称的Git版本库或在Git库中建立特别的分支实现对主页的维护。

3.5.1. 创建个人主页

GitHub 为每一个用户分配了一个二级域名.github.io，用户为自己的二级域名创建主页很容易，只要在托管空间下创建一个名为.github.io的版本库，向其master分支提交网站静态页面即可，其中网站首页为index.html。下面以gotgithub用户为例介绍如何创建个人主页。

- 用户gotgithub创建一个名为gotgithub.github.io的Git版本库。

  在GitHub上创建版本库的操作，参见“第3.1节 创建新项目”。

- 在本地克隆新建立的版本库。

```
$ git clone git@github.com:gotgithub/gotgithub.github.io.git
$ cd gotgithub.github.io/
```

- 在版本库根目录中创建文件index.html作为首页。

```
$ printf "<h1>GotGitHub's HomePage</h1>It works.\n" > index.html
```

- 建立提交。

```
$ git add index.html
$ git commit -m "Homepage test version."
```

- 推送到GitHub，完成远程版本库创建。

```
$ git push origin master
```

- 访问网址： http://gotgithub.github.io/ 。

  最多等待10分钟，GitHub就可以完成新网站的部署。网站完成部署后版本库的所有者会收到邮件通知。

  还有要注意访问用户二级域名的主页要使用HTTP协议非HTTPS协议。

3.5.2. 创建项目主页

如前所述，GitHub会为每个账号分配一个二级域名.github.io作为用户的首页地址。实际上还可以为每个项目设置主页，项目主页也通过此二级域名进行访问。

例如gotgithub用户创建的helloworld项目如果启用了项目主页，则可通过网址 http://gotgithub.github.io/helloworld/访问。

为项目启用项目主页很简单，只需要在项目版本库中创建一个名为gh-pages的分支，并向其中添加静态网页即可。也就是说如果项目的Git版本库中包含了名为gh-pages分支的话，则表明该项目提供静态网页构成的主页，可以通过网址http://.github.io/访问到。

下面以用户gotgithub的项目helloworld为例，介绍如何维护项目主页。

如果本地尚未从GitHub克隆helloworld版本库，执行如下命令。

```
$ git clone git@github.com:gotgithub/helloworld.git
$ cd helloworld
```

当前版本库只有一个名为master的分支，如果直接从master分支创建gh-pages分支操作非常简单，但是作为保存网页的gh-pages分支中的内容和master分支中的可能完全不同。如果不希望gh-pages分支继承master分支的历史和文件，即想要创建一个干净的gh-pages分支，需要一点小技巧。

若使用命令行创建干净的gh-pages分支，可以从下面三个方法任选一种。

第一种方法用到两个Git底层命令：git write-tree和git commit-tree。步骤如下：

- 基于master分支建立分支gh-pages。

```
$ git checkout -b gh-pages
```

- 删除暂存区文件，即相当于清空暂存区。

```
$ rm .git/index
```

- 创建项目首页index.html。

```
$ printf "hello world.\n" > index.html
```

- 添加文件index.html到暂存区。

```
$ git add index.html
```

- 用Git底层命令创建新的根提交，并将分支gh-pages重置。

```
$ git reset --hard $(echo "branch gh-pages init." | git commit-tree $(git write-tree))
```

- 执行推送命令，在GitHub远程版本库创建分支gh-pages。

```
$ git push -u origin gh-pages
```

第二种方法用到Git底层命令：git symbolic-ref。步骤如下：

- 用git symbolic-ref命令将当前工作分支由master切换到一个尚不存在的分支gh-pages。

```
$ git symbolic-ref HEAD refs/heads/gh-pages
```

- 删除暂存区文件，即相当于清空暂存区。

```
$ rm .git/index
```

- 创建项目首页index.html。

```
$ printf "hello world.\n" > index.html
```

- 添加文件index.html到暂存区。

```
$ git add index.html
```

- 执行提交。提交完毕分支gh-pages完成创建。

```
$ git commit -m "branch gh-pages init."
```

- 执行推送命令，在GitHub远程版本库创建分支gh-pages。

```
$ git push -u origin gh-pages
```

第三种方法没有使用任何Git底层命令，是从另外的版本库获取提交建立分支。操作如下：

- 在helloworld版本库之外创建另外一个版本库，例如helloworld-web。

```
$ git init ../helloworld-web
$ cd ../helloworld-web
```

- 在helloworld-web版本库中创建主页文件index.html。

```
$ printf "hello world.\n" > index.html
```

- 添加文件index.html到暂存区。

```
$ git add index.html
```

- 执行提交。

  实际提交到master分支，虽然提交说明中出现的是gh-pages 。

```
$ git commit -m "branch gh-pages init."
```

- 切换到helloworld版本库目录。

```
$ cd ../helloworld
```

- 从helloworld-web版本库获取提交，并据此创建gh-pages分支。

```
$ git fetch ../helloworld-web
$ git checkout -b gh-pages FETCH_HEAD
```

- 执行推送命令，在GitHub远程版本库创建分支gh-pages。

```
$ git push -u origin gh-pages
```

无论哪种方法，一旦在GitHub远程版本库中创建分支gh-pages，项目的主页就已经建立。稍后（不超过10分钟），用浏览器访问下面的地址即可看到刚刚提交的项目首页：http://gotgithub.github.io/helloworld/ 。

除了以上通过命令行创建gh-pages分支为项目设定主页之外，GitHub还提供了图形操作界面。如图3-19所示。

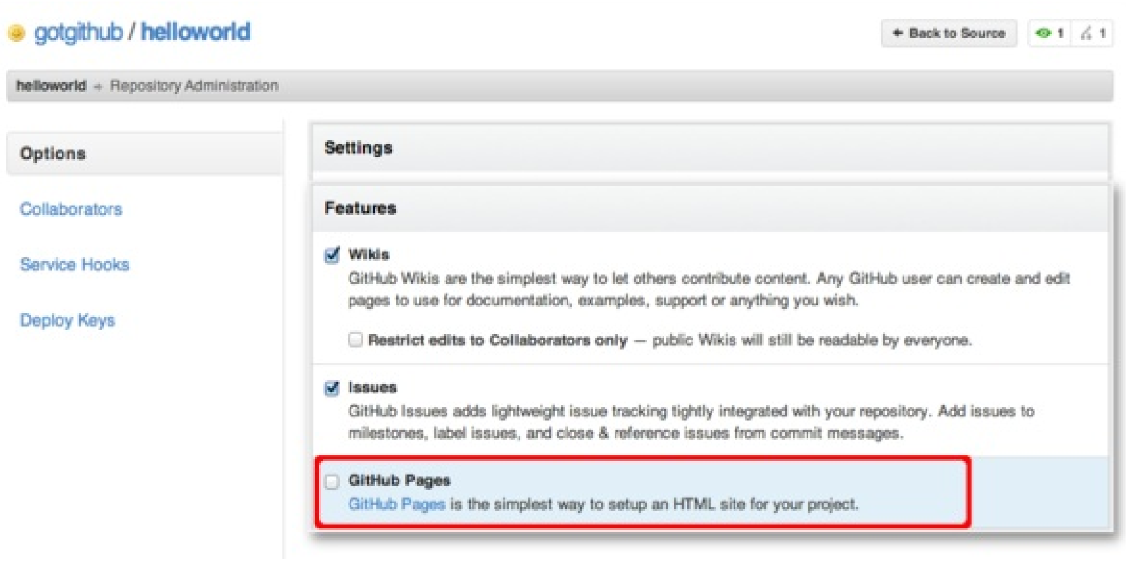


图3-19：项目管理页面中的GitHub Pages选项

当在项目管理页面中勾选“GitHub Pages”选项，并按照提示操作，会自动在项目版本库中创建gh-pages分支。然后执行下面命令从版本库检出gh-pages分支，对项目主页进行相应定制。

```
$ git fetch
$ git checkout gh-pages
```

3.5.3. 使用专有域名

无论是用户主页还是项目主页，除了使用github.com下的二级域名访问之外，还可以使用专有域名。实现起来也非常简单，只要在master分支（用户主页所在版本库）或gh-pages分支（项目版本库）的根目录下检入一个名为CNAME的文件，内容为相应的专有域名。当然还要更改专有域名的域名解析，使得该专有域名的IP地址指向相应的GitHub二级域名的IP地址。

例如worldhello.net(“Hello, world”最为程序员所熟知，2002年申请不到helloworld相关域名便退而求其次，申请了 worldhello.net。)是我的个人网站，若计划将网站改为由GitHub托管，并由账号gotgit通过个人主页提供服务，可做如下操作。

首先按照前面章节介绍的步骤，为账号gotgit设置账户主页。

1. 在账户gotgit下创建版本库gotgit.github.io以维护该账号主页。

   地址： https://github.com/gotgit/gotgit.github.io/

2. 将网站内容提交并推送到该版本库master分支中。

   即在gotgit.github.io版本库的根目录下至少包含一个首页文件，如index.html。还可以使用下节将要介绍到的 Jekyll 技术，让网页有统一的显示风格，此时首页文件可能并非一个完整的HTML文档，而是套用了页面模版。

3. 至此当访问网址http://gotgit.github.io时，会将账号gotgit的版本库gotgit.github.io中的内容作为网站内容显示出来。

接下来进行如下操作，使得该网站能够使用专有域名www.worldhello.net提供服务。

1. 在账号gotgit的版本库gotgit.github.io根目录下添加文件CNAME，文件内容为：www.worldhello.net。

   参见： https://github.com/gotgit/gotgit.github.io/blob/master/CNAME

2. 然后更改域名www.worldhello.net的IP地址，指向域名gotgit.github.io对应的IP地址（注意不是github.com的IP地址）。

   完成域名的DNS指向后，可试着用ping或dig命令确认域名www.worldhello.net和gotgit.github.io指向同一IP地址。

```
$ dig @8.8.8.8 -t a www.worldhello.net
...
; ANSWER SECTION:
www.worldhello.net.     81078   IN      A       204.232.175.78

$ dig @8.8.8.8 -t a gotgit.github.io
...
; ANSWER SECTION:
gotgit.github.io.      43200   IN      A       204.232.175.78
```

设置完成后用浏览器访问 http://www.worldhello.net/ 即可看到由账号gotgit的版本库gotgit.github.io维护的页面。若将域名worldhello.net（不带www前缀）也指向IP地址204.232.175.78，则访问网址http://worldhello.net/会发现GitHub体贴地将该网址重定向到正确的地址http://www.worldhello.net/。

在账号gotgit下的其他版本库，若包含了gh-pages分支，亦可由域名www.worldhello.net访问到。

- 网址 http://www.worldhello.net/doc 实际对应于版本库 gotgit/doc 。
- 网址 http://www.worldhello.net/gotgit 实际对应于版本库 gotgit/gotgit 。
- 网址 http://www.worldhello.net/gotgithub 实际对应于版本库gotgit/gotgithub 。

3.5.4. 使用Jekyll维护网站

Jekyll是一个支持Textile、Markdown等标记语言的静态网站生成软件，还支持博客和网页模版，由Tom Preston-Werner（GitHub创始人之一）开发。Jekyll用Ruby语言实现，项目在GitHub的托管地址： http://github.com/mojombo/jekyll/ ，专有的URL地址为：http://jekyllrb.com/。

GitHub为用户账号或项目提供主页服务，会从相应版本库的master分支或gh-pages分支检出网页文件，然后执行 Jekyll 相应的命令对网页进行编译。因此在设计GitHub的用户主页和项目主页时都可以利用Jekyll，实现用Markdown等标记语言撰写网页及博客，并用页面模版实现网页风格的统一。

安装Jekyll最简单的方法是通过RubyGems安装，会自动将Jekyll依赖的directory_watcher、liquid、open4、maruku和classifier等Gem包一并安装。

```
$ gem install jekyll
```

如果安装过程因编译扩展模组遇到错误，可能是因为尚未安装所需的头文件，需要进行如下操作：

- 对于Debian Linux、Ubuntu等可以用如下方法安装所需软件包：

```
$ sudo apt-get install ruby1.8-dev
```

- 如果是Red Hat、CentOS或Fedora等系统，使用如下命令安装：

```
$ sudo yum install ruby-devel
```

- 对于Mac OSX，可能需要更新RubyGems，如下：

```
$ sudo gem update --system
```

Jekyll安装完毕，执行下面的命令显示软件版本：

```
$ jekyll -v
Jekyll 0.11.0
```

要学习如何用Jekyll设计网站，可以先看一下作者Tom Preston-Werner在GitHub上的个人网站是如何用Jekyll制作出来的。

克隆版本库：

```
$ git clone git://github.com/mojombo/mojombo.github.com.git
```

版本库包含的文件如下：

```
$ cd mojombo.github.com
$ ls -F
CNAME           _config.yml     _posts/         css/            index.html
README.textile  _layouts/       atom.xml        images/         random/
```

版本库根目录下的index.html文件不是一个普通的HTML文件，而是使用Liquid模版语言[2]定义的页面。

```
1 ---
 2 layout: default
 3 title: Tom Preston-Werner
 4 ---
 5
 6 <div id="home">
 7   <h1>Blog Posts</h1>
 8   <ul class="posts">
 9     {% for post in site.posts %}
10       <li><span>{{ post.date | date_to_string }}</span> &raquo; <a href="{{ post.url }
11     {% endfor %}
12   </ul>
   ...
63 </div>
```

为方便描述为内容添加了行号，说明如下：

- 第1-4行是YAML格式的文件头，设定了该文件所使用的模版文件及模版中要用到的变量。

  凡是设置有YAML文件头的文件（目录_layouts除外）无论文件扩展名是什么，都会在Jekyll编译时进行转换。若源文件由Markdown等标记语言撰写（扩展名为.md、.textile等），Jekyll还会将编译后的文件还将以扩展名.html来保存。

- 其中第2行含义为使用default模版。

  对应的模版文件为_layouts/default.html。

- 第3行设定本页面的标题。

  在模版文件_layouts/default.html中用{{ page.title }}语法嵌入所设置的标题。下面是模版文件中部分内容：

```
<head>
   <meta http-equiv="content-type" content="text/html; charset=utf-8" />
   <title>{{ page.title }}</title>
```

- 第6行开始的内容绝大多数是标准的HTML语法，其中夹杂少量Liquid模版特有的语法。
- 第9行和第11行，对于有着Liquid或其他模版编程经验的用户，不难理解其中出现的由“{%”和“%}”标识的指令是一个循环指令（for循环），用于逐条对博客进行相关操作。
- 第10行中由“{{”和“}}”标识的表达式则用于显示博文的日期、链接和标题。

非下划线（ `_` ）开头的文件（包括子目录中文件），如果包含YAML文件头，就会使用Jekyll进行编译，并将编译后的文件复制到目标文件夹（默认为 `_site` 目录）下。对于包含YAML文件头并用标记语言Markdown等撰写的文件，还会将编译后的文件以 `.html` 扩展名保存。而以

下划线开头的文件和目录有的直接忽略不予处理（如 `_layouts` 、 `_site` 目录等），有的则需要特殊处理（如 `_post` 目录）。

目录_post用于保存博客条目，每个博客条目都以 `<YYYY>-<MM>-<DD>-<blog-tiltle>` 格式的文件名命名。扩展名为.md的为Markdown格式，扩展名为.textile的为Textile格式。这些文件都包含类似的文件头：

```
---
layout: post
title: How I Turned Down $300,000 from Microsoft to go Full-Time on GitHub
---
```

即博客使用文件 `_layouts/post.html` 作为页面模版，而不是index.html等文件所使用的 `_layouts/default.html` 模版。这些模版文件都采用Liquid模版语法。保存于_post目录下的博客文件编译后会以 `<YYYY>/<MM>/<DD>/<blog-title>.html` 文件名保存在输出目录中。

在根目录下还有一个配置文件_config.yml用于覆盖Jekyll的默认设置，例如本版本库中的设置。

```
markdown: rdiscount
pygments: true
```

第1行设置使用rdiscount软件包作为Markdown的解析引擎，而非默认的Maruku。第2行开启pygments支持。对于中文用户强烈建议通过配置文件_config.yml重设 markdown 解析引擎，默认的 Maruku 对中文支持不好，而使用 rdiscount 或 kramdown 均可。关于该配置文件的更多参数详见Jekyll项目维基[3] 。

编译Jekyll编辑网站只需在根目录执行jekyll命令，下面的命令是GitHub更新网站所使用的默认指令。

```
$ jekyll --pygments --safe
```

现在执行这条命令，就会将整个网站创建在目录_site下。

如果没有安装Apache等Web服务器，还可以使用Jekyll的内置Web服务器。

```
$ jekyll --server
```

默认在端口4000开启Web服务器。

网址 http://gitready.com/ 是一个提供Git使用小窍门的网站，如图3-20所示。

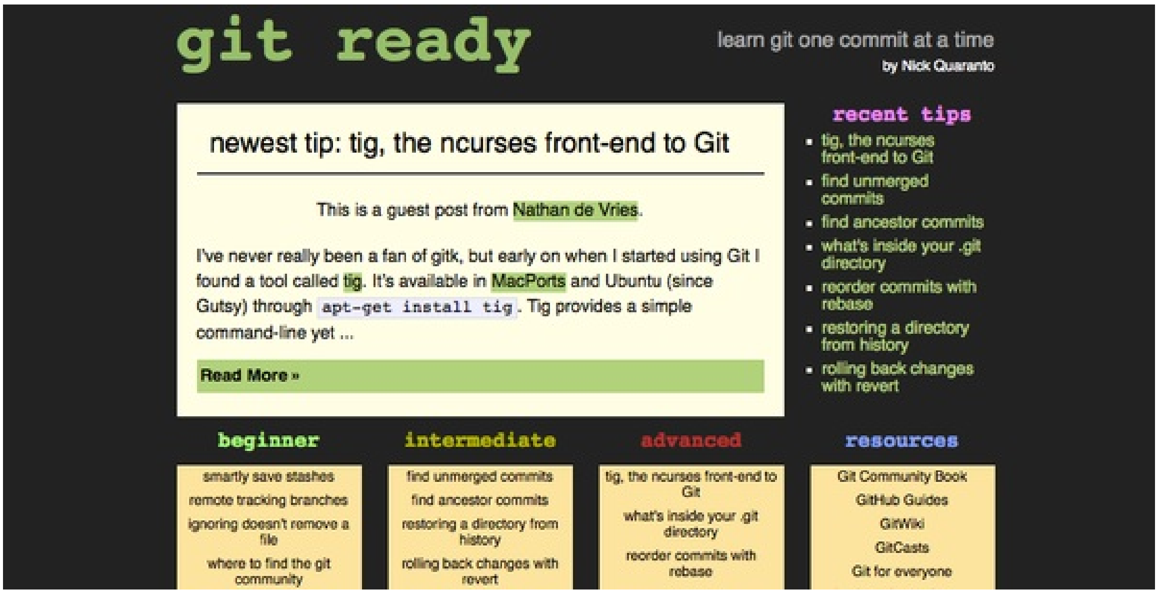


图3-20：Git Ready 网站

你相信这是一个用Jekyll制作的网站么？看看该网站对应的IP，会发现其指向的正是GitHub。研究GitHub上 gitready 用户托管的版本库，会发现en版本库的gh-pages分支负责生成gitready.com网站，de版本库的gh-pages分支负责生成德文网站de.gitready.com，等等。而gitready版本库则是各种语种网站的汇总。

我的个人网站也使用Jekyll构建并托管在GitHub上，网址：http://www.worldhello.net/ 。

4. 工作协同

项目落户GitHub后，一定希望有越来越多的人能参与其中。GitHub提供了包括传统的问题追踪系统、维基，还包括了分布式版本控制系统特有的协同工具。

4.1. Fork + Pull模式

参与GitHub中的项目开发，最常用和推荐的首选方式是“Fork + Pull”模式。在“Fork + Pull”模式下，项目参与者不必向项目创建者申请提交权限，而是在自己的托管空间下建立项目的派生（Fork）。

如果一个开源项目派生出另外的项目，通常意味着项目的分裂和开发团队的削弱，而GitHub中的项目派生则不会，而且正好相反，GitHub中的项目派生是项目壮大的体现。所有的派生项目都会有链接指向原始项目，派生项目没有独立的缺陷追踪系统（ISSUE），而是必须利用创建者本人的项目中的缺陷追踪系统。至于在派生项目中创建的提交，可以非常方便地利用GitHub的Pull Request工具向原始项目的维护者发送Pull Request。

下面以GotGit版本库为例，介绍如何利用GitHub提供的Fork和Pull Request工具实现工作协同。

4.1.1. 版本库派生

GotGit版本库[1]用于维护《Git权威指南》一书的官网和勘误，下面演示的勘误表修改是由王胜[2]通过GitHub之外的一个缺陷追踪平台报告的[3]。他在报告中，甚至直接用GNU diff格式告诉我该如何修改。

下面就以用户gotgithub身份，访问版本库 https://github.com/gotgit/gotgit/ ，添加新的勘误。如图4-1所示，gotgit项目在之前的示例中已经被我们关注但尚未Fork。



图4-1：原gotgit项目

点击项目名称右侧的Fork按钮，便在gotgithub用户自己的托管空间下创建项目派生，派生项目版本库出现在版本库列表中，如图4-2。

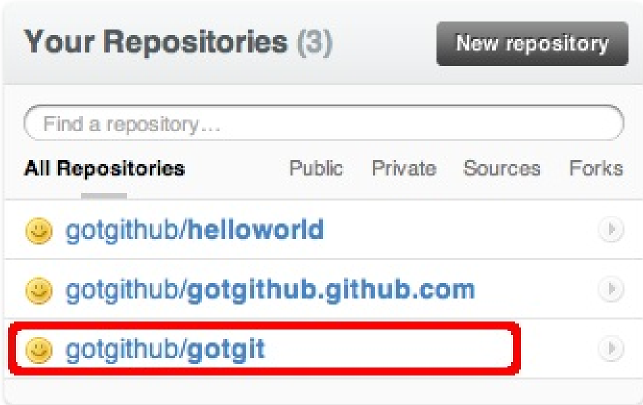


图4-2：gotgithub用户的项目列表

访问派生后的版本库，会发现和派生前的几乎相同，除了没有缺陷跟踪（ISSUE），以及标识了该项目派生之前的原路径等。如图4-3所示。

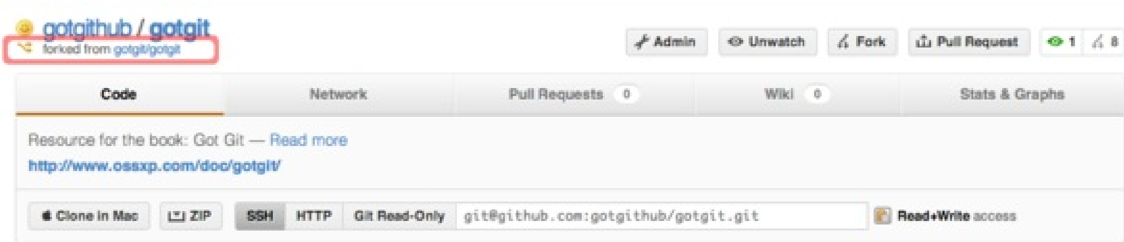


图4-3：派生的gotgit项目

现在gotgithub用户就在本地派生的版本库中提交。

- 克隆 gotgithub/gotgit 版本库。

```
$ git clone git@github.com:gotgithub/gotgit.git
$ cd gotgit
```

- 为了向问题的发现者致敬，并经王胜同意，以他的身份进行提交。

```
$ git config user.name "Wang Sheng"
$ git config user.email wangsheng@ossxp.com
```

- 编辑errata.mkd文件(版本库 gotgit/gotgit 已将勘误文件重命名为errata.md。)，录入新发现的书中的文字错误。

```
$ vi errata.mkd
```

- 对error.mkd的改动如下：

```
$ git diff
diff --git a/errata.mkd b/errata.mkd
index b0b68fb..29e40cf 100644
--- a/errata.mkd
+++ b/errata.mkd
@@ -14,5 +14,6 @@
 |      66 | 倒数第11行                        | Author（提交者）              |  Author（作者）
 |     144 | 第1行                             | \`$ **git rev-parse  A^{tree}  A:**  | $ **git rev
 |     218 | 第8行                             | 况下，Gits标识出合并冲突，             | 况下，Git标识出合
+|     369 | 第21行                            | 但 `-i` 参数仅当对一个项执行时才有效。 | 但 `-i` 参数仅当
 |     516 | 倒数第15行                        | **oldtag="cat"**                | **oldtag=\`cat\`**
```

- 提交修改。至于提交说明中出现的编号，是为了和缺陷跟踪系统关联，会在后面章节介绍。

```
$ git add -u
$ git commit -m "Fixed #3: should be 项目, not 项."
```

- 推送提交到GitHub。

```
$ git push
```

访问GitHub上的派生项目页面，会看到以用户whangsheng在master分支[5]创建的提交。如图4-4所示。

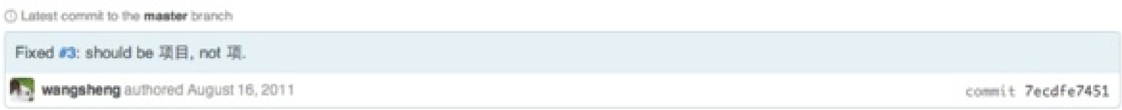


图4-4：派生版本库中的新提交

4.1.2. Pull Request

那么如何能够让gotgit原始项目的创建者知道这个派生项目及新的提交呢？GitHub提供的工具就是“Pull Request”。注意到图4-3右上方“Pull Request”按钮了么？点击该按钮进入Pull Request创建界面。

在弹出的Pull Request创建界面中，点击菜单中的“Commits”，查看所包含的提交。如图4-5所示。



图4-5：Pull Request包含的提交

点击菜单中的“Files Changed”，查看所包含的提交。如图4-6所示。



图4-6：Pull Request包含的改动差异

点击菜单中的“Preview Discussion”，填写Pull Request的标题和内容，完成Pull Request的创建。如图4-7所示。

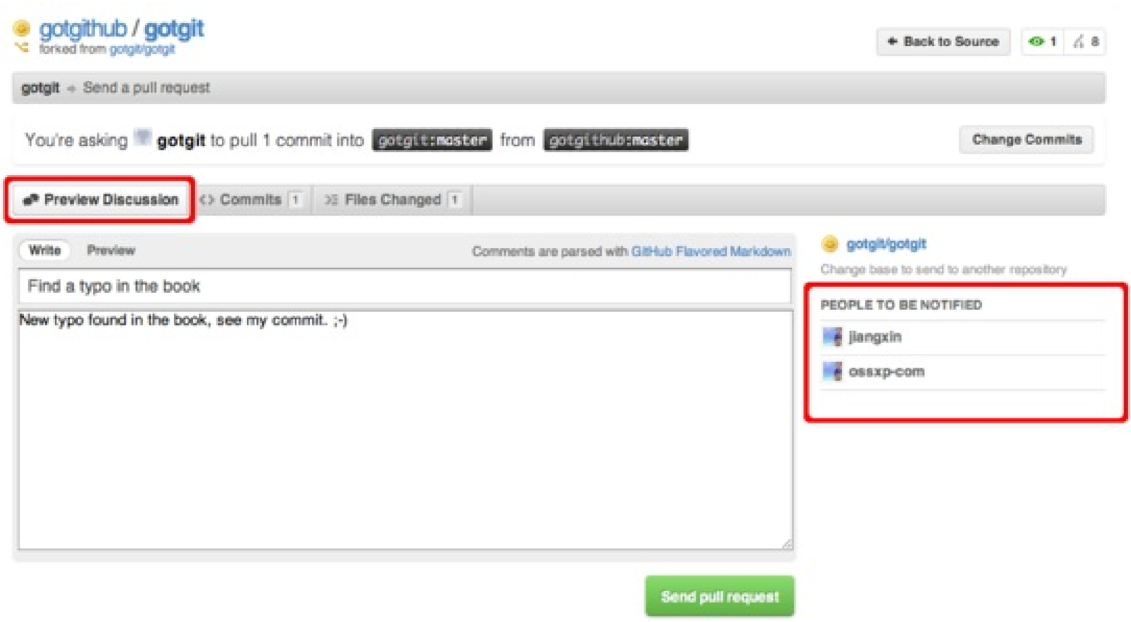


图4-7：Pull Request的提交界面

当Pull Request发出后，项目gotgit的开发者会收到通知邮件，如图4-8所示。



图4-8：Pull Request的通知邮件

点击邮件中的URL链接，以项目gotgit的开发者（如ossxp-com）身份登录，看到如图4-9的视图。之所以看到有两个用户参与到此Pull Request，是因为Pull Request创建者和提交的作者是不同的用户。图4-9下方的表单可以向Pull Request追加评论，或者关闭此Pull Request。

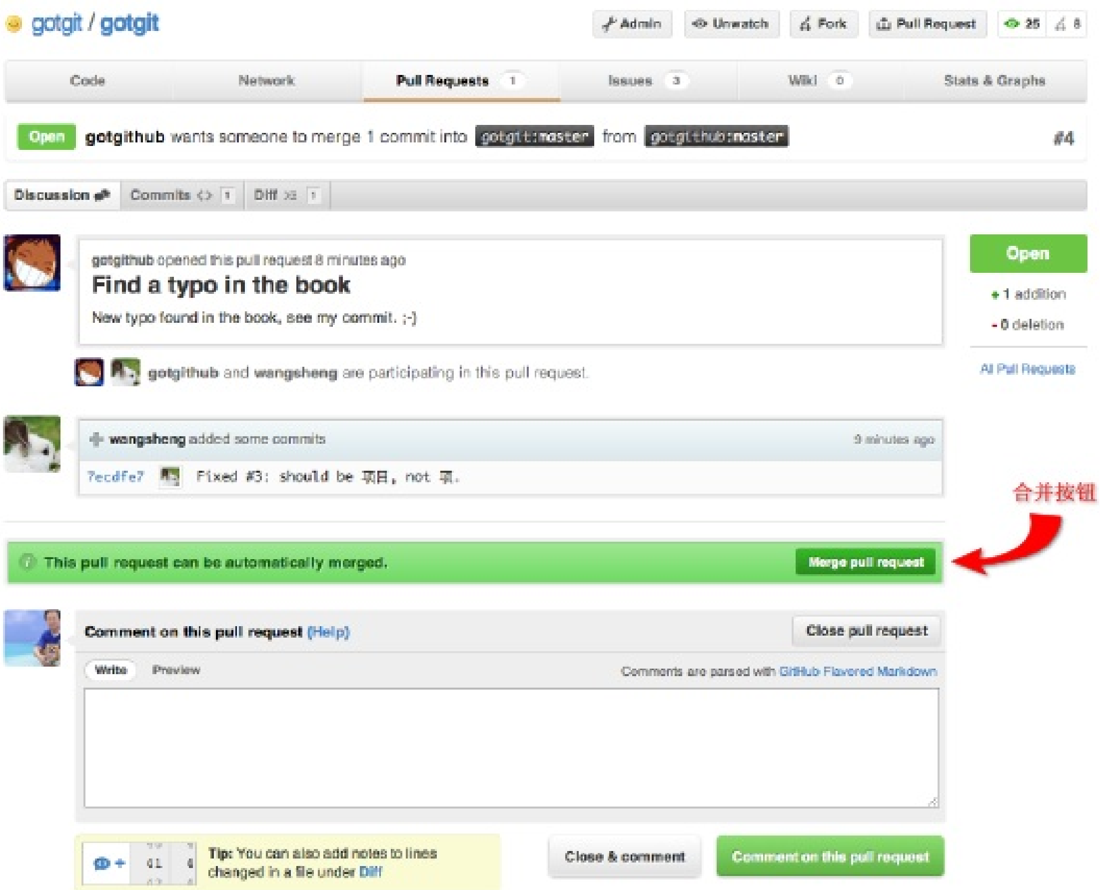


图4-9：Pull Request接收者视图

GitHub如果检测到Pull Request中包含的提交可以直接合并，会显示自动合并的提示信息，点击图4-9中提示信息中的自动合并按钮，显示图4-10的自动合并对话框。

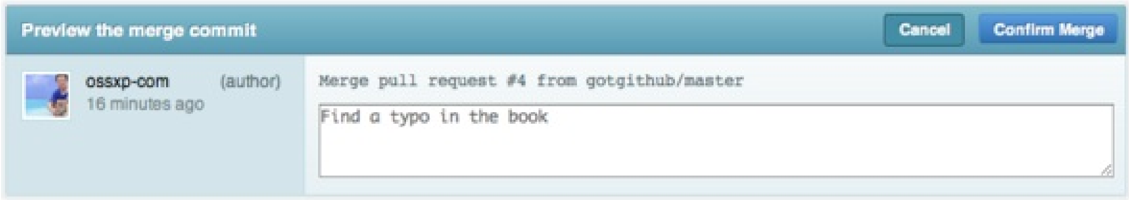


图4-10：Pull Request的通知邮件

点击“Confirm Merge”按钮即完成Pull Request中所含提交的自动合并。自动合并完成后，Pull Request页面下方会以评论的形式出现相关提示，并自动关闭Pull Request。如图4-11所示。



图4-11：Pull Request关闭

4.1.3. 手工合并

Pull Request提供的自动合并显示在提交日志中是什么样子的呢？以用户ossxp-com身份检出版本库，会看到用户wangsheng的提交已经合并到版本库中。

```
$ git clone git@github.com:gotgit/gotgit.git
$ cd gotgit
$ git log --graph -3
*   commit 6c1f1ee152629fd2f8d00ebe92c27a32d068d756
|\  Merge: 00c6c4b 7ecdfe7
| | Author: OpenSourceXpress
| | Date:   Tue Aug 16 01:23:47 2011 -0700
| |
| |     Merge pull request #4 from gotgithub/master
| |
| |     Find a typo in the book
| |
| * commit 7ecdfe7451412cfb2e65bb47c12cf2162e21c841
|/  Author: Wang Sheng
|   Date:   Tue Aug 16 10:17:53 2011 +0800
|
|       Fixed #3: should be 项目, not 项.
|
* commit 00c6c4bfab9824bd967440902ce87440f9e87852
| Author: Jiang Xin
| Date:   Wed Aug 3 11:50:31 2011 +0800
|
|     Change font color for stronger text from red to brown.
```

可以看出GitHub的自动合并产生了一个合并提交，类似执行git merge --no-ff命令。也就是说即使用户wangsheng的提交是一个“快进式提交”（基于gotgit/gotgit版本库最新提交所做的提交），也要产生一个合并提交。

可能有人并不喜欢这种用--no-ff参数的非标准的合并方式，因为这种合并产生了一个多余的提交，可能增加代码评审的负担。若要取消GitHub的自动合并也很简单，因为Git无所不能：

```
$ git reset --hard HEAD^  # 回退一个提交，即回退到当前提交的第一个父提交
$ git rev-parse HEAD      # 检查是否正确的回退
00c6c4bfab9824bd967440902ce87440f9e87852
$ git push -f             # 强制推送回退的 master 分支
```

下面就演示一下当收到他人的Pull Request后，该如何手动合并。实际上在很多情况下，Pull Request所含提交有可能造成合并冲突，那样的话GitHub不再、也不能提供自动合并功能，就必须采用手工合并的方式。

- 将Pull Request发出者的派生版本库添加为一个新的源。

  例如收到来自gotgithub用户的Pull Request，不妨以gotgithub为名添加新的源。

```
$ git remote add gotgithub https://github.com/gotgithub/gotgit.git
```

- 此时版本库中有两个源，一个克隆时自动建立的origin，另外一个就是新增加的gotgithub。

```
$ git remote -v
gotgithub       https://github.com/gotgithub/gotgit.git (fetch)
gotgithub       https://github.com/gotgithub/gotgit.git (push)
origin  git@github.com:gotgit/gotgit.git (fetch)
origin  git@github.com:gotgit/gotgit.git (push)
```

- 获取远程版本库gotgithub的分支和提交。

```
$ git fetch gotgithub
From https://github.com/gotgithub/gotgit
 * [new branch]      gh-pages   -> gotgithub/gh-pages
 * [new branch]      master     -> gotgithub/master
```

- 现在除了本地分支master外，还有若干远程分支，如下：

```
$ git branch -a
* master
  remotes/gotgithub/gh-pages
  remotes/gotgithub/master
  remotes/origin/HEAD -> origin/master
  remotes/origin/gh-pages
  remotes/origin/master
```

- 将远程分支remotes/gotgithub/master（可简写为gotgithub/master）合并到当前分支中。

```
$ git merge gotgithub/master
Updating 00c6c4b..7ecdfe7
Fast-forward
 errata.mkd |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 查看提交说明，看到此次合并没有产生不必要的合并提交。

```
$ git log --graph -2
* commit 7ecdfe7451412cfb2e65bb47c12cf2162e21c841
| Author: Wang Sheng
| Date:   Tue Aug 16 10:17:53 2011 +0800
|
|     Fixed #3: should be 项目, not 项.
|
* commit 00c6c4bfab9824bd967440902ce87440f9e87852
| Author: Jiang Xin
| Date:   Wed Aug 3 11:50:31 2011 +0800
|
|     Change font color for stronger text from red to brown.
```

- 将合并推送到GitHub版本库中。

```
$ git push
```

4.1.4. 在线编辑

GitHub提供了在线编辑功能，这样可以无需克隆版本库、无需使用Git即可完成对版本库中文件的修改，甚至可以在你的iPad甚至iPhone上完成对文件的修改。

以gotgithub账户身份登录GitHub，访问之前派生而来的版本库gotgithub/gotgit中的文件，例如文件errata.md(版本库 gotgit/gotgit 已重构。分支gh-pages中文件errata.md文件来自于原master分支的errata.mkd文件，地址：https://github.com/gotgithub/gotgit/blob/gh-pages/errata.md)，会看到其中一个“Edit this file”的按钮，如图4-12所示。

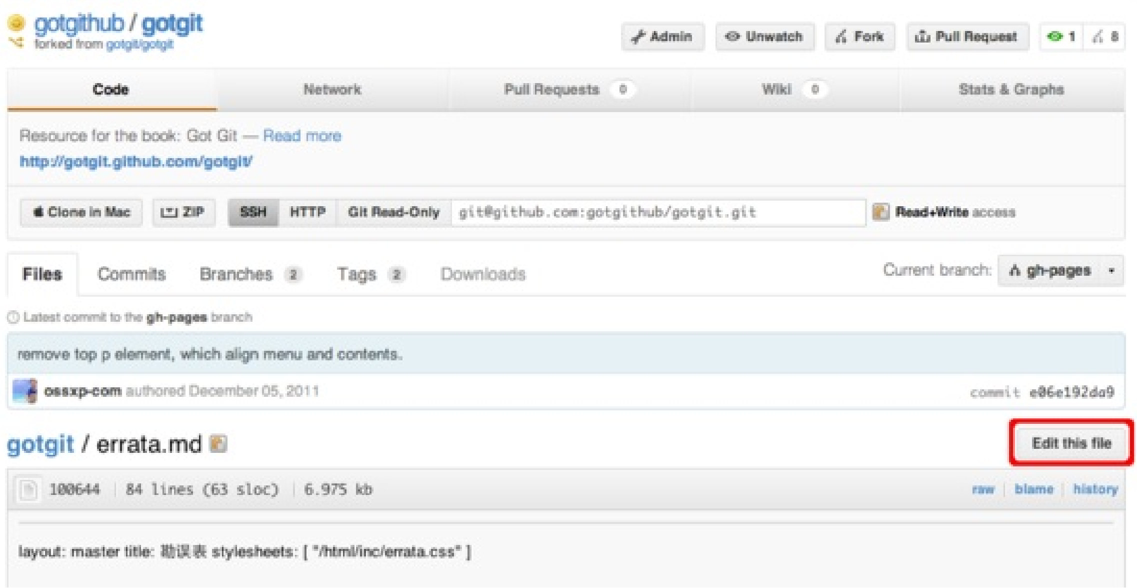


图4-12：浏览自己版本库中文件

点击图4-12中的“Edit this file”按钮，开始在线编辑文件errata.md，编辑器还支持语法加亮，如图4-13所示。

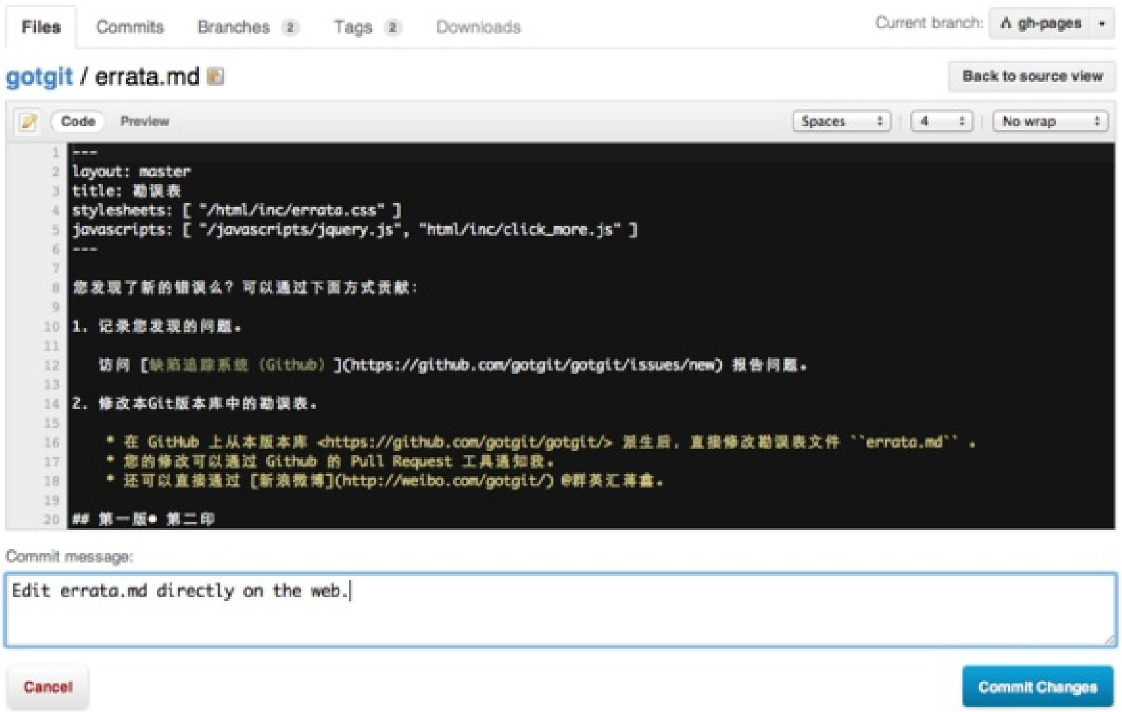


图4-13：编辑文件

4.1.5. 简化的 Fork + Pull Request

到目前，我们已经了解了GitHub的三大武器：Fork、Pull Request和在线编辑。对于最常用的“Fork + Pull Request”操作，GitHub还提供了一个快捷模式。即GitHub对于无权更改的他人版本库中的文件，提供了一个类似在线编辑的按钮，名为“Fork and edit this file”按钮，自动完成版本库派生和在线编辑，即将三大武器一勺烩。

访问他人版本库（尚未在自己空间派生）中的文件，例如访问下面地址：http://git.io/hello-world-makefile(即地址 https://github.com/ossxp-com/hello-world/blob/master/src/Makefile)。显示他人（ossxp-com）版本库hello-world中的src/Makefile文件，如图4-14所示。

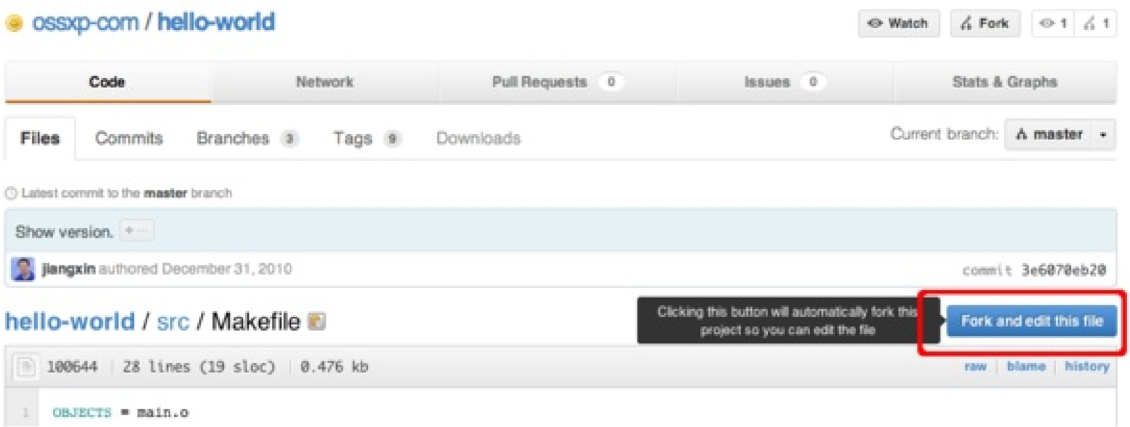


图4-14：浏览他人版本库中文件

点击图4-14中的“Fork and edit this file”按钮，会自动在自己托管空间创建派生版本库，并开始在线编辑文件src/Makefile，如图4-15所示。

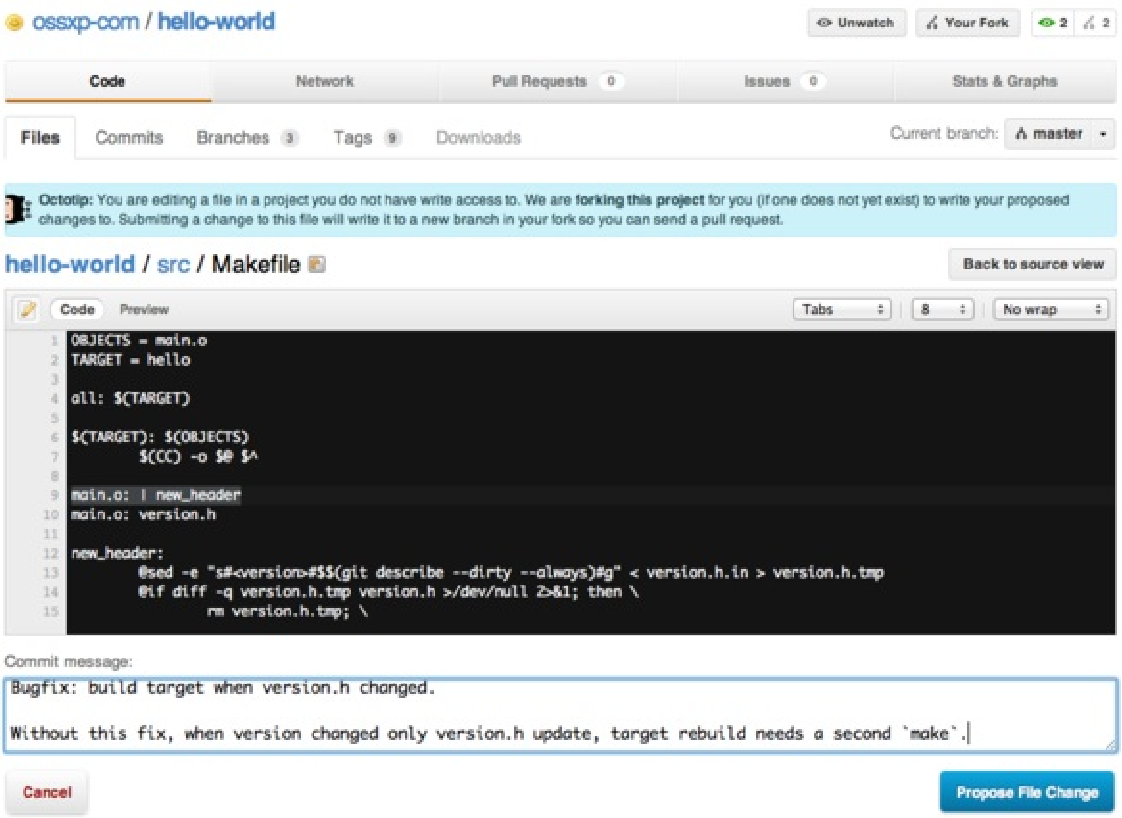


图4-15：派生并编辑文件

文件修改完毕，点击“Propose File Change”按钮，会将改动作提交到派生的版本库中，并马上开启一个新的Pull Request。如图4-16所示。

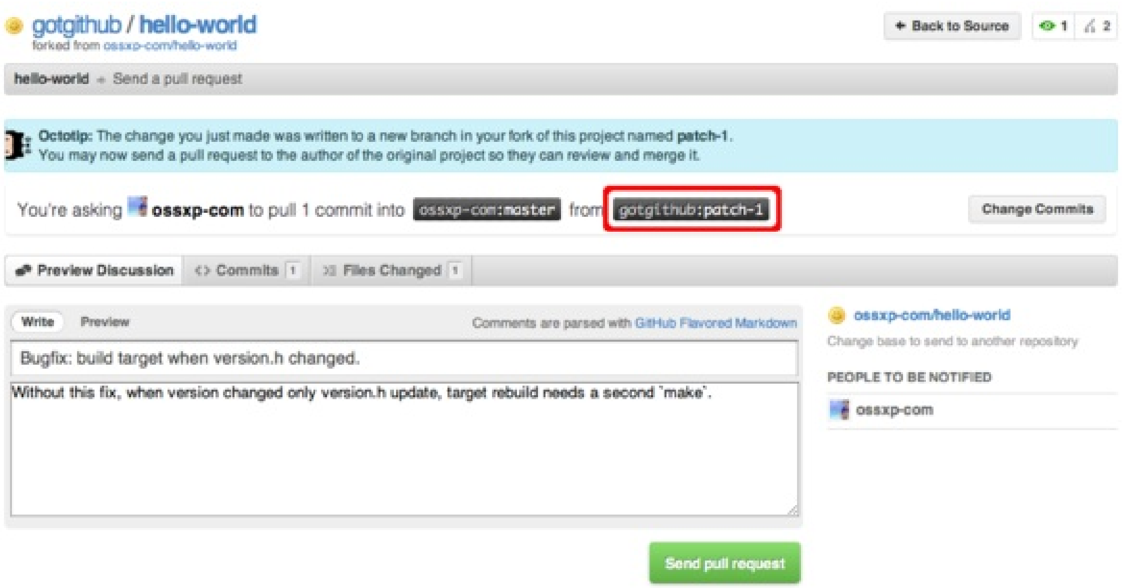


图4-16：编辑完毕自动开启Pull Request

点击“Send pull request”按钮完成Pull Request的创建。如果仔细查看图4-16，会发现Pull Request所包含的修改发生在gotgithub/hello-world派生版本库中的patch-1分支中，并非通常的master分支。

原版本库ossxp-com/hello-world的开发者会收到一封邮件，通知有新的Pull Request，如下所示（前四行为信头）：

```
From: GotGitHub
Date: 2011/12/17
Subject: [hello-world] Bugfix: build target when version.h changed.  (#1)
To: Jiang Xin

Without this fix, when version changed only version.h update, target rebuild needs a seco

You can merge this Pull Request by running:

 git pull https://github.com/gotgithub/hello-world patch-1

Or you can view, comment on it, or merge it online at:

 https://github.com/ossxp-com/hello-world/pull/1

-- Commit Summary --

* Bugfix: build target when version.h changed.

-- File Changes --

M src/Makefile (3)

-- Patch Links --

 https://github.com/ossxp-com/hello-world/pull/1.patch
 https://github.com/ossxp-com/hello-world/pull/1.diff

---
Reply to this email directly or view it on GitHub:
https://github.com/ossxp-com/hello-world/pull/1
```

版本库ossxp-com/hello-world的管理员既可以通过GitHub提供的图形化界面完成对 Pull Request 的审核和合并，也可以在命令行下完成。正如邮件中所述若使用命令行，操作如下：

```
$ git pull https://github.com/gotgithub/hello-world patch-1
```

4.2. 共享版本库

除了独具特色的“Fork + Pull”的分布式工作模式，GitHub同样支持传统的集中式协同工作模式。

4.2.1. 版本库授权

GitHub可以通过多种途径为版本库授权，让版本库成为多人共享的版本库，从而让项目管理者围绕项目创建一个核心开发团队。下面以gotgithub账号下的helloworld版本库为例，介绍如何设置版本库的多人共享。

进入gotgithub/helloworld项目的管理界面，点击左侧导航条中的“Collaborators”，可以查看及添加项目的合作者，如图4-17所示。

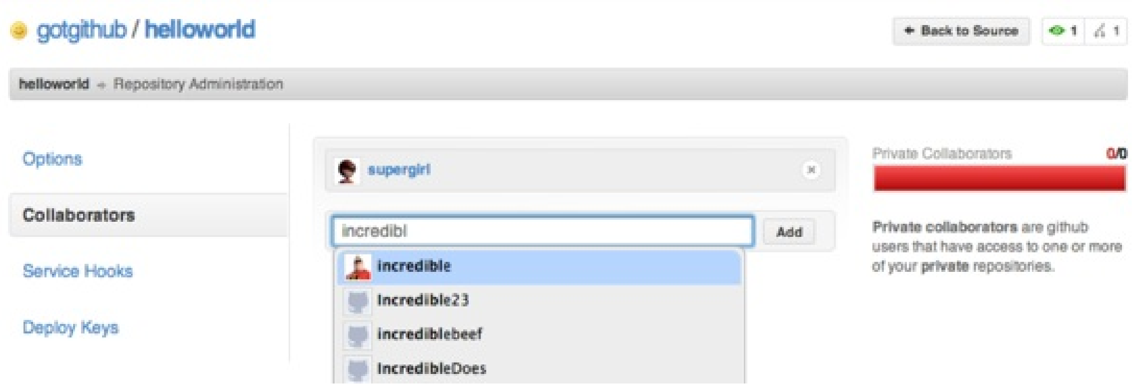


图4-17：添加项目合作者

图4-17为项目添加了两个合作者：supergirl和incredible。添加到项目当中的合作者会收到通知邮件，告知已经被加入到相应的项目当中。当新加入的项目合作者，如incredible，登录GitHub，在仪表板的版本库列表中会看到源自gotgithub用户的版本库，如图4-18所示。



图4-18：合作者项目列表

从图4-18可以看出GitHub用户incredible自己创建的项目托管在自己的空间下，而作为合作者参与的项目仍然托管在原创建者（gotgithub）的空间下，这一点明显和派生（Fork）而来的项目不同。图4-19是以incredible登录GitHub访问gotgithub/helloworld的界面。

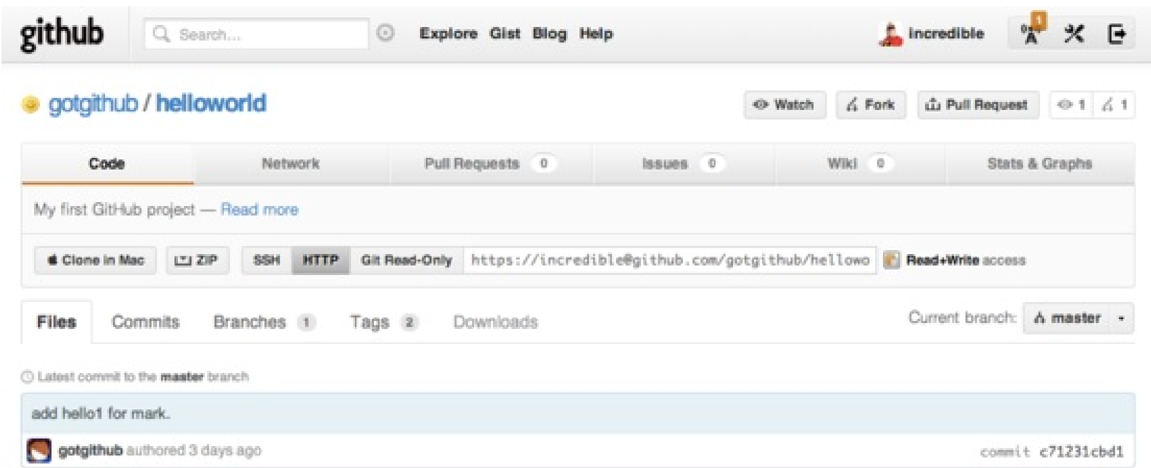


图4-19：以合作者身份访问项目

用户incredible对版本库gotgithub/helloworld拥有写入权限。和gotgithub用户稍有区别的是没有管理员权限。

4.2.2. 与传统集中式工作模式的异同

传统的集中式版本控制系统，如CVS、SVN，所有用户都访问同一个版本库。采用集中式工作模式的GitHub用户也同样是访问同一版本库。

对于由用户gotgithub创建的helloworld版本库，添加了合作者supergirl和incredible，三个人克隆版本库使用如下命令。

- 用户 gotgithub 克隆版本库。

```
gotgithub$ git clone https://gotgithub@github.com/gotgithub/helloworld.git
```

- 用户 supergirl 克隆版本库。

```
supergirl$ git clone https://supergirl@github.com/gotgithub/helloworld.git
```

- 用户 incredible 克隆版本库。

```
incredible$ git clone https://incredible@github.com/gotgithub/helloworld.git
```

传统集中式版本控制系统，所有的提交历史数据都在唯一的版本库中，各个提交是顺序进行的。为了防止多用户在提交时相互覆盖，集中式版本控制系统发明了很多方法 。有的采用编辑锁的形式（如VSS），更改文件前对文件锁定，其他人禁止对文件进行访问（甚至无法读取），完成修改并提交后，文件解锁。有的如SVN，允许多人同时编辑同一文件，但只有先进行提交的才能成功，后提交的会遇到“过时”错误，必须先获取版本库中的新增提交并和本地修改进行合并。即传统集中式版本控制系统，提交时必须和唯一的版本库所在的服务器保持连接，而且提交有可能会失败。

对于像Git这样的分布式版本控制系统，提交总是会成功，这是因为提交并不涉及和共享服务器的交互，是针对本地克隆版本库进行的本地操作。采用集中式的工作模式，共享版本库作为各个用户各自本地版本库数据交换、沟通的中介，只有在本地克隆版本库需要和共享版本库同步的时候才要和服务器建立连接。例如将本地所做的一个或多个提交推送到共享服务器，或者将服务器上新的提交获取到本地克隆版本库。

实际上无论采用分布式还是集中式的工作模式，Git都好像工作在一个独立的分支上（克隆即分支），即使共享版本库和本地克隆版本库的分支名都叫做master。如图4-20，三个用户克隆gitgithub/helloworld版本库后，各自在本地执行了一次或多次提交。

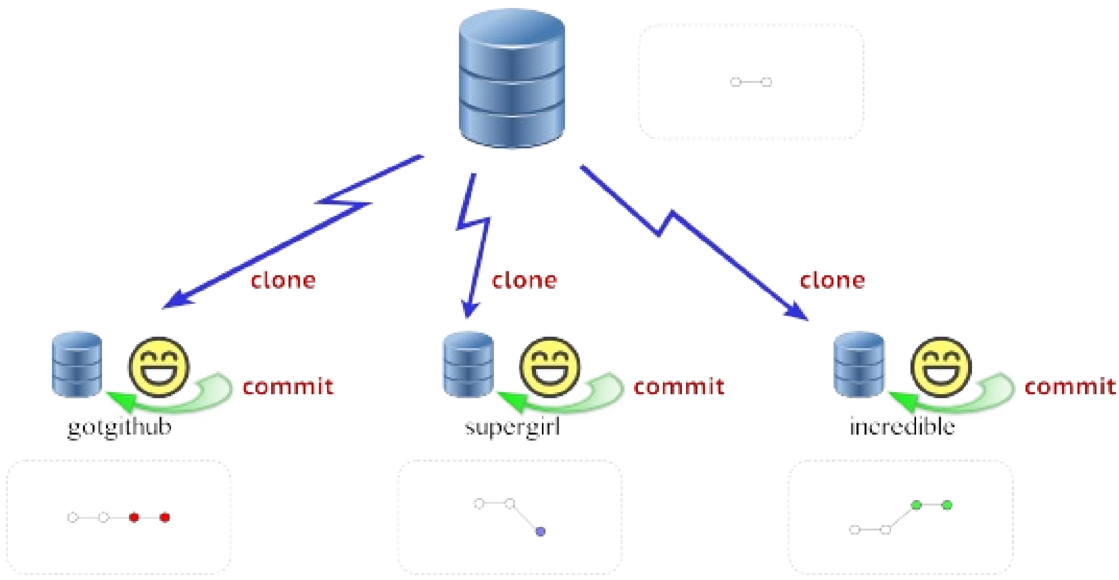


图4-20：在本地版本库中的提交

三个用户各自的提交都会非常顺利，但是一旦决定将本地提交推送到共享服务器时就可能遇到麻烦。用户 gotgithub 先执行推送，会非常顺利。如图4-21所示。

```
gotgithub$ git push
```

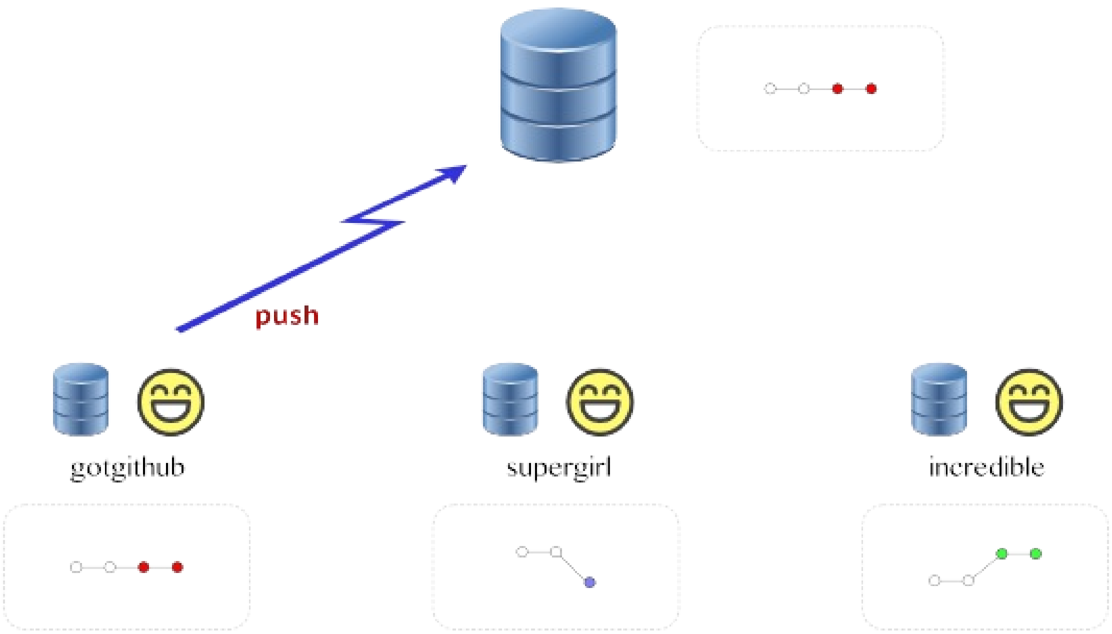


图4-21：用户gotgithub完成推送

而其他人就没有这么幸运了，会报告错误：遇到非快进式推送。Git的推送操作就像SVN等集中式版本控制系统的提交操作那样，先执行者成功，后执行者糟糕（要解决冲突，自动或手动）。

4.2.3. 合并后推送

当用户gotgithub完成推送后，共享版本库以及三个用户的本地版本库如图4-21所示。其中共享版本库变得和gotgithub用户的本地版本库相一致。此时如果用户supergirl执行推送，会遇到错误：非快进式推送。

```
supergirl$ git push
To https://supergirl@github.com/gotgithub/helloworld.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://supergirl@github.com/gotgithub/helloworld.git
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

GitHub并不对强制推送进行限制，但是用户supergirl不要用git push -f命令强制推送，因为那样会覆盖掉共享版本库中用户gotgithub的推送，正确的做法是获取共享版本库中新提交，并在本地版本库中和本地提交合并。即执行：

```
supergirl$ git fetch
supergirl$ git merge
```

获取和合并操作过程如图4-22所示。

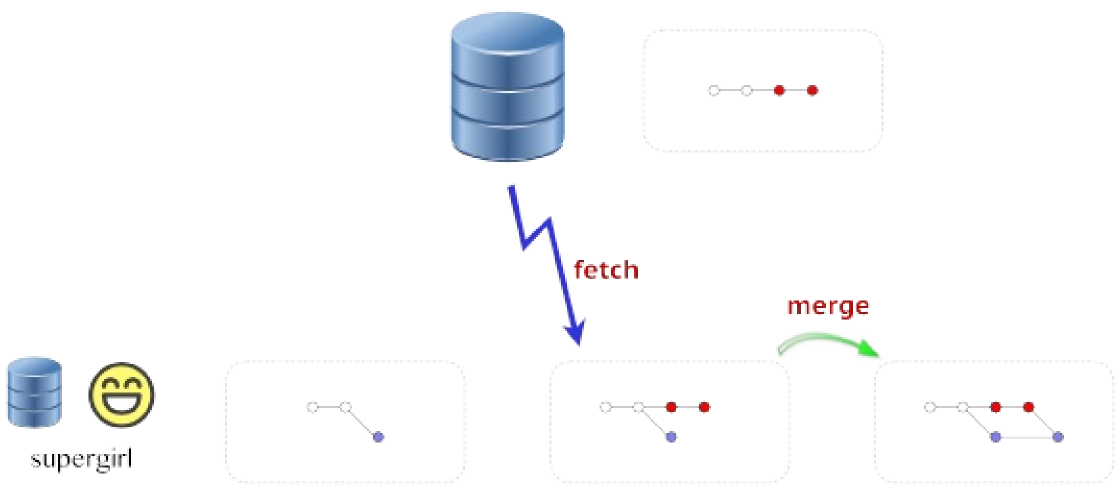


图4-22：合并操作示意图

实际上用户supergirl只需执行一条命令便可完成所有的操作：

```
supergirl$ git pull
```

即：git pull = git fetch + git merge。

但是合并操作并不总是会成功，如果自动合并失败，会在暂存区对合并前后文件进行标识，工作区进入冲突解决状态，在冲突解决完成之前不能提交。Git支持多种图形工具帮助完成冲突解决，执行如下命令，即可自动调用已安装的冲突解决工具。

```
supergirl$ git mergetool
```

冲突解决完毕，执行提交即完成冲突解决。如果在冲突解决过程把本地文件搞得一团糟，随时可以取消合并操作。执行命令git reset --hard会取消冲突的合并让本地版本库回到合并之前的状态。

成功完成合并后将本地版本库中的提交推送到共享版本库：

```
supergirl$ git push
```

完成推送后的版本库示意图如图4-23所示。

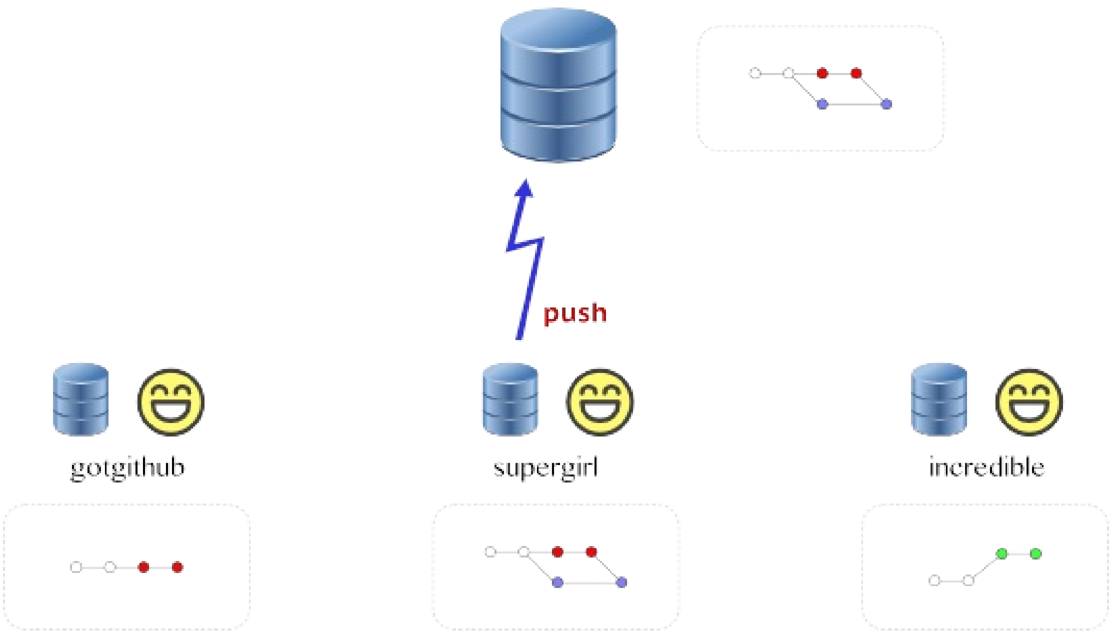


图4-23：完成合并后推送

4.2.4. 合并还是变基

合并并非多个开发者的工作成果融合的唯一选择，有时甚至并非最佳选择。一方面合并会产生除了合并双方（或多方）所有提交外的一个新提交，增加了代码审核的负担，另一方面本地多个提交混杂一起与远程分支合并会更困难。在特定情况下，变基是合并之外的另一个选择。

图4-24展示用户incredible采用合并和变基两种不同解决方案的操作结果。图中右上是合并操作后的结果，右下是变基操作后的结果。

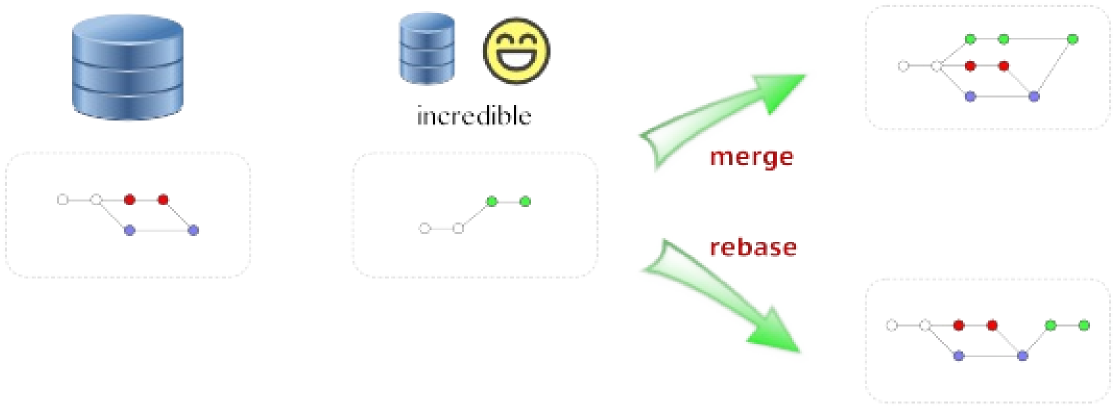


图4-24：合并和变基结果比较

若用户 incredible 选择变基操作，执行命令如下：

- 获取远程版本库的提交到本地的远程分支。

```
incredible$ git fetch origin
```

- 执行变基操作，将本地master分支的提交变基到新的远程分支中。

```
incredible$ git rebase origin/master
```

如果一切顺利，变基后推送到共享版本库。

```
incredible$ git push
```

推送后的版本库状态如图4-25所示。

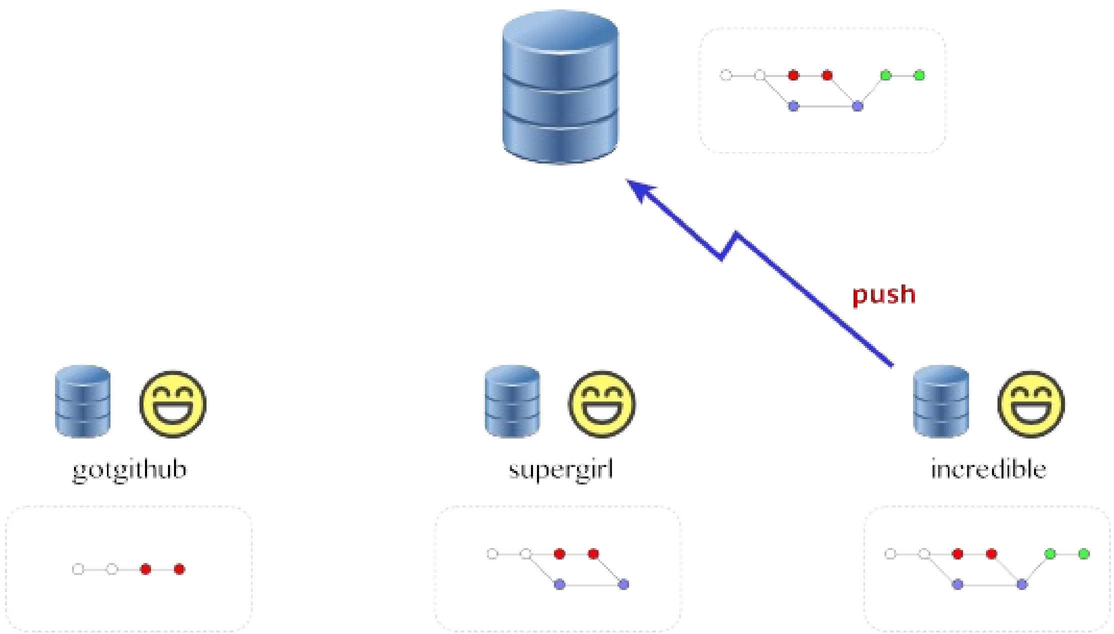


图4-25：变基后推送

如果希望在执行git pull时自动使用git rebase取代默认的git merge操作，可以在git pull命令行添加参数--rebase如下：

```
$ git pull --rebase
```

或者通过配置变量设置当前分支使用变基策略，即每次执行git pull命令时对于master分支，采用变基操作取代默认的合并操作。

```
$ git config branch.master.rebase true
```

如果希望本地所有克隆版本库在执行git pull时都改变默认行为，将变基作为首选，则如下设置全局变量。

```
$ git config --global branch.autosetuprebase true
```

4.3. 组织和团队

GitHub 在早期没有专门为组织提供账号，很多企业用户或大型开源组织只好使用普通用户账号作为组织的共享账号来使用。后来，GitHub推出了组织这一新的账号管理模式，满足大型开发团队的需要。

- 组织账号是不能用来登录的，它包含一个Owner（拥有者）用户组，只有属于这个组的用户在登录后，才能切换为组织的管理者。
- 可以创建任意多的团队（Team）即角色，对属于组织的用户进行管理。Owner Team就是组织中权限最高的角色。
- 组织和用户一样可以创建项目，但是组织没有SSH公钥配置，也不能以组织的身份操作版本库。
- 组织没有工作描述之类的个人账号才拥有的属性。

4.3.1. 创建新组织

组织是非登录账号，不能像创建普通登录账号那样直接创建，而是需要以GitHub用户身份登录，然后再创建自己的组织，创建者成为组织天然的管理者。

图4-26就是用户gotgithub登录后，通过点击右上角的账号设置图标进入账号设置界面，然后再点击菜单中的“Organizations”进入组织管理界面。

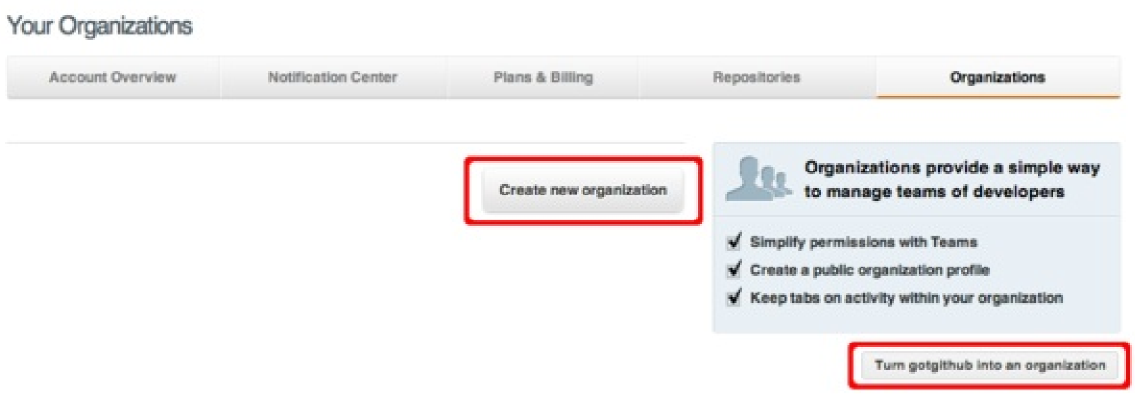


图4-26：账号设置中的组织管理

在初始的组织管理界面中组织列表为空，即尚不属于任何组织。可以选择把当前用户gotgithub的账号转换为一个组织账号（前提是gotgithub的账号不属于任何组织）。提供这一账号迁移功能是因为在GitHub提供组织这一新功能之前，很多公司或团队以个人身份创建GitHub账号，但是以组织的形象出现，对于这类账号，GitHub提供了由个人账号向组织账号迁移的途径。

在这里我们不进行这一迁移，而是以用户gotgithub的身份创建一个新的组织。点击“Create New Organization”按钮，显示创建组织表单，如图4-27所示。

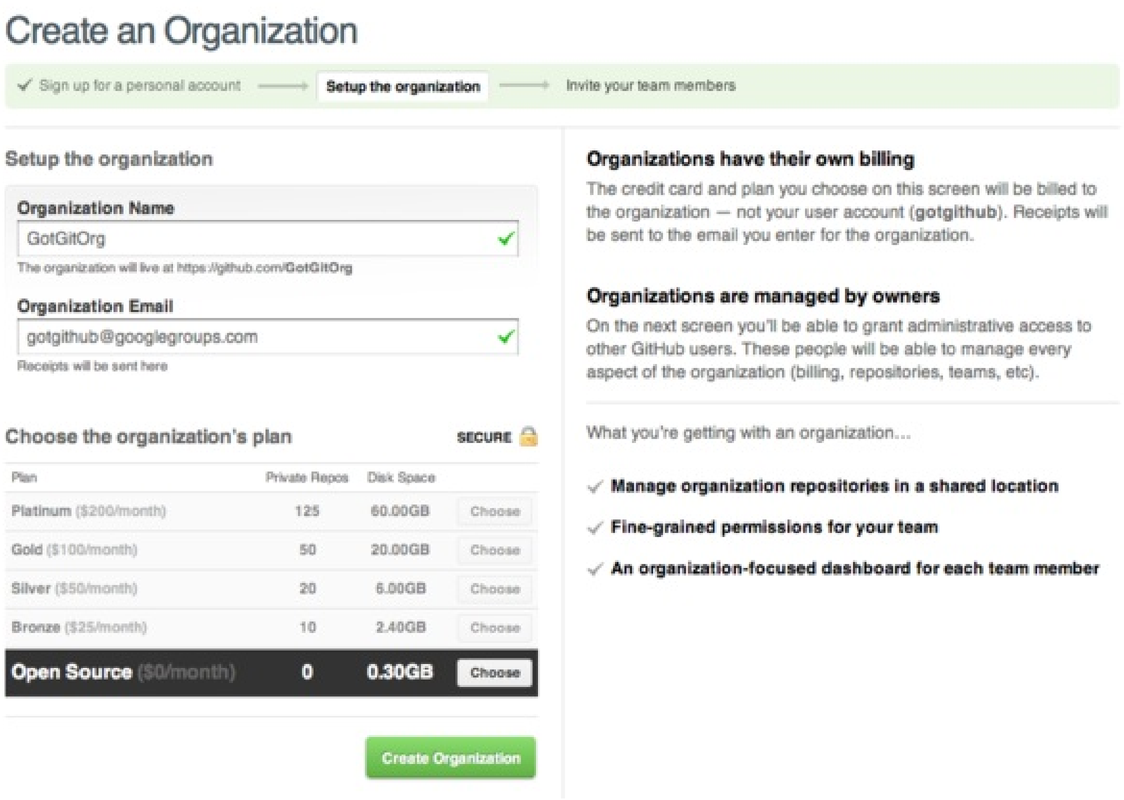


图4-27：创建新组织

这里填写组织名为 GotGitOrg。创建组织还要选择一个付费方案，默认会选择免费的没有私有版本库的开源方案。

接下来为新建组织设定拥有者（Owner），如图4-28所示。当前用户，即正在创建组织的用户，理所当然成为组织拥有者之一。还可以为组织指派更多的组织拥有者，多个组织拥有者的权限并无差别，都可以管理组织，甚至可以将其他用户从拥有者团队中删除。

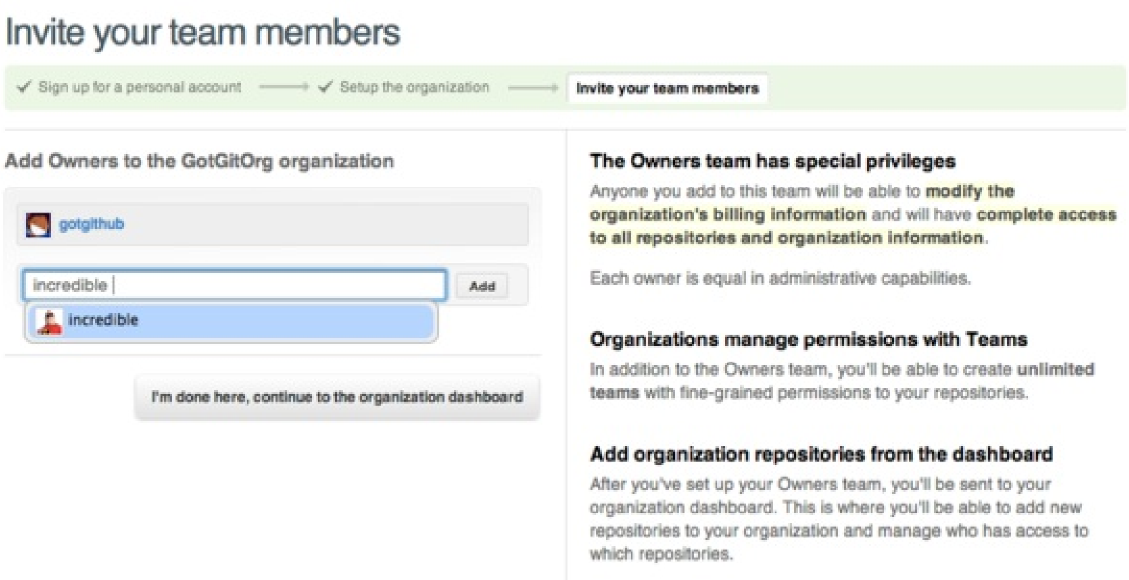


图4-28：指派组织拥有者

完成创建后，访问用户账号设置界面中的组织面板，如图4-29所示，列出当前用户所属的组织（GotGitOrg）。可以重新对组织进行设定，或者退出组织。注意因为当前用户已经属于一个以上的组织，所以右侧将当前用户转换为组织的按钮被置灰。

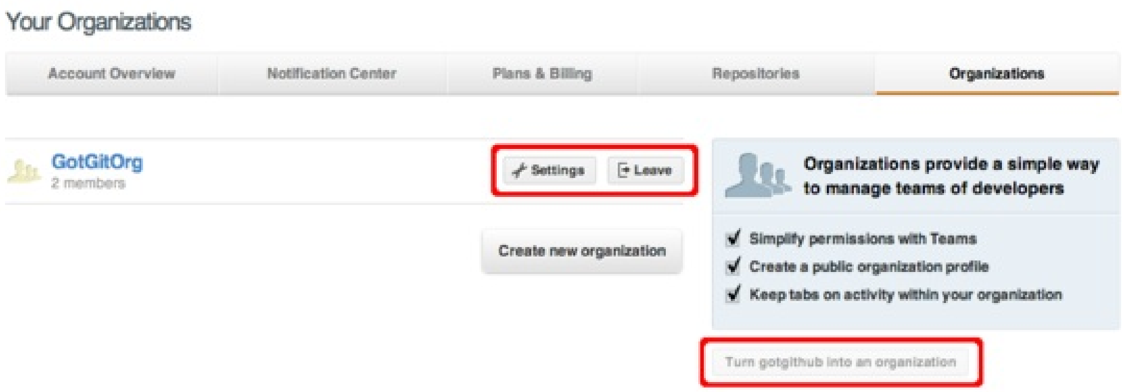


图4-29：加入组织后的组织管理界面

4.3.2. 组织管理

当用户gotgithub成为新建组织GotGitOrg的一员后，就可以在用户和组织的界面之间切换。点击页面左上角“github”文字图标进入仪表板界面。



图4-30：用户仪表板界面

仪表板页面左上角“github”文字图标的下面就是用户上下文列表框。点击用户上下文下拉列表，如图4-31所示。

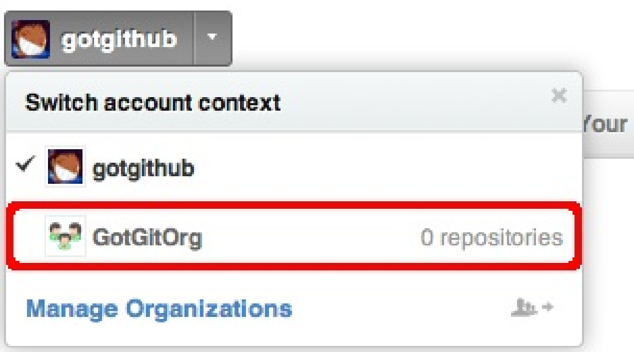


图4-31：用户上下文切换列表

在用户上下文列表中选择组织GotGitOrg作为用户上下文后，则仪表板中显示的菜单和个人账号仪表板菜单略有不同，如图4-32所示。

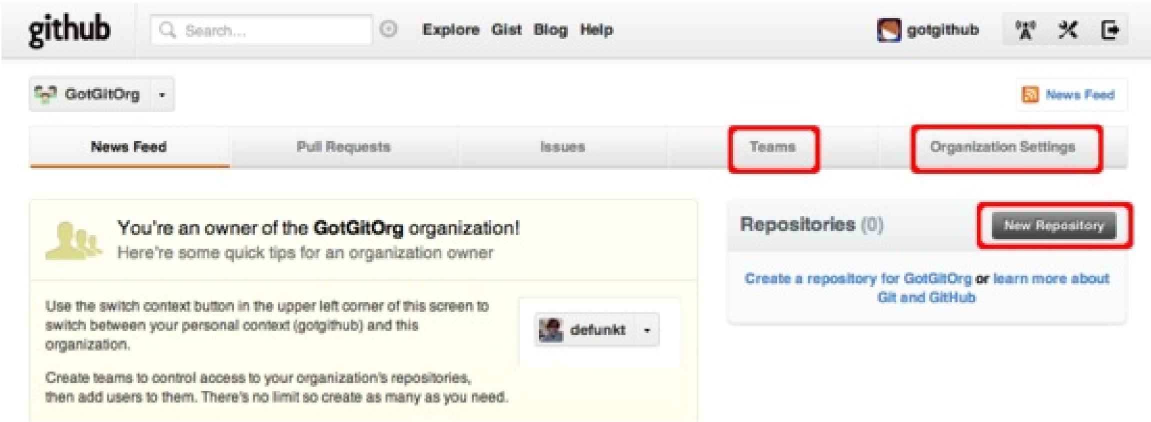


图4-32：组织GotGitOrg的仪表板界面

组织的仪表板界面与用户仪表板的不同之处在于增加了团队管理（Team）和组织管理（Organization Settings）。选择菜单中的“Team”进入团队管理界面，可以在组织中添加任意数量的团队。添加新团队的界面如图4-33所示。

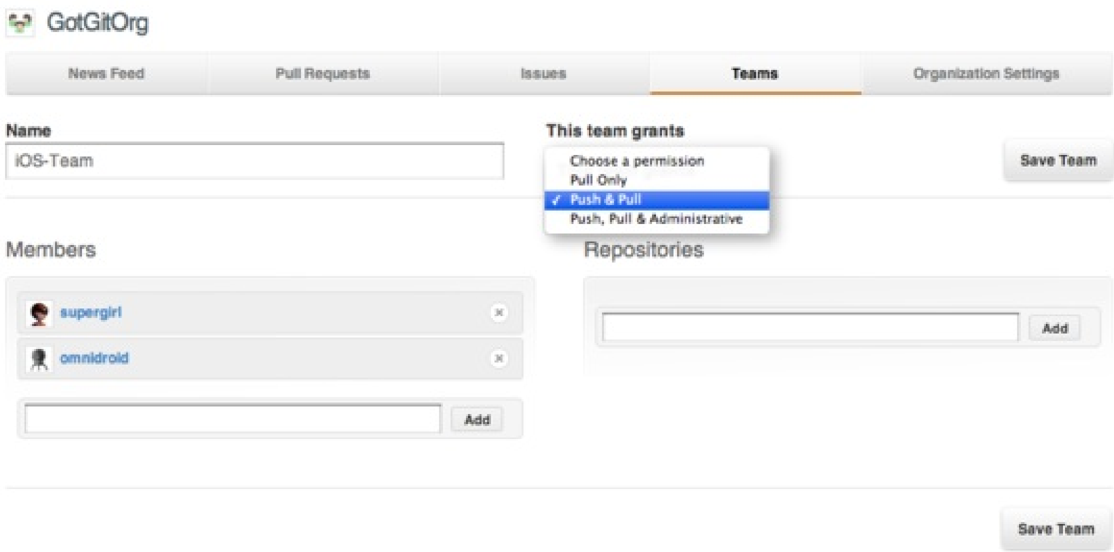


图4-33：添加新团队iOS-Team

创建一个团队需要提供四个选项（如图4-33）：

1. 团队名称。如：iOS-Team。
2. 团队成员。如：supergirl和omnidroid两个用户作为团队成员。
3. 团队权限。有三个选择：只读（Pull Only）、读写（Push & Pull）、读写并管理（Push, Pull & Administrative）。
4. 授权版本库。可以添加一个或多个版本库，只有对授权的版本库才拥有指定权限。

其中团队授权中的只读授权对于免费组织账号创建的开源项目没有实际意义，因为开源项目人人可读，只有对于付费的组织账号创建的私密版本库才体现出价值。关于付费账号和私密版本库将在后面的章节介绍。接下来介绍如何在组织账号下创建版本库。

4.3.3. 版本库管理

组织拥有独立的项目托管空间，点击页面左上角的“github”文字图标进入组织账号的仪表板界面。刚刚建立的组织账号的版本库尚未创建，点击图4-32所示的“New Repository”按钮，创建版本库（即项目）。

新建版本库的界面如图4-34所示。

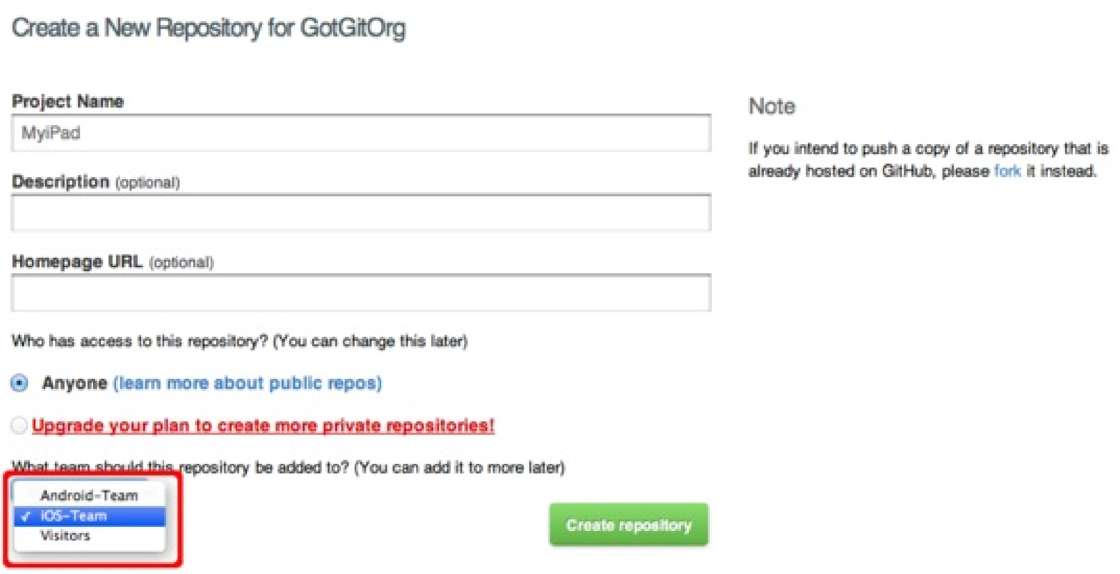


图4-34：新建项目界面

在组织的托管空间创建项目与在普通用户的空间下创建稍有不同，增加了团队设置下拉框。图4-34显示在创建名为MyiPad项目时，只能为项目指派一个已定义团队，要想为项目指派更多团队可以在项目创建完毕通过项目管理界面添加。

下面来看一看如何为已建立项目指派更多的团队。进入项目管理页面，点击左侧菜单项“Team”显示项目的团队管理界面，可以通过该界面，为项目添加和移除团队，如图4-35所示。

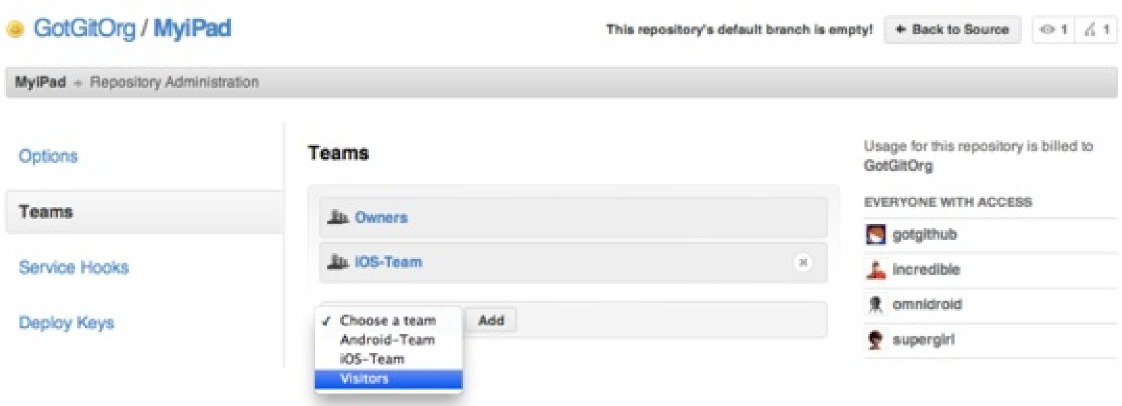


图4-35：项目的团队管理

属于团队的项目（版本库）可以转移给个人，反之亦然。图4-36展示了如何通过项目管理界面在用户和组织之间转移项目（版本库）。

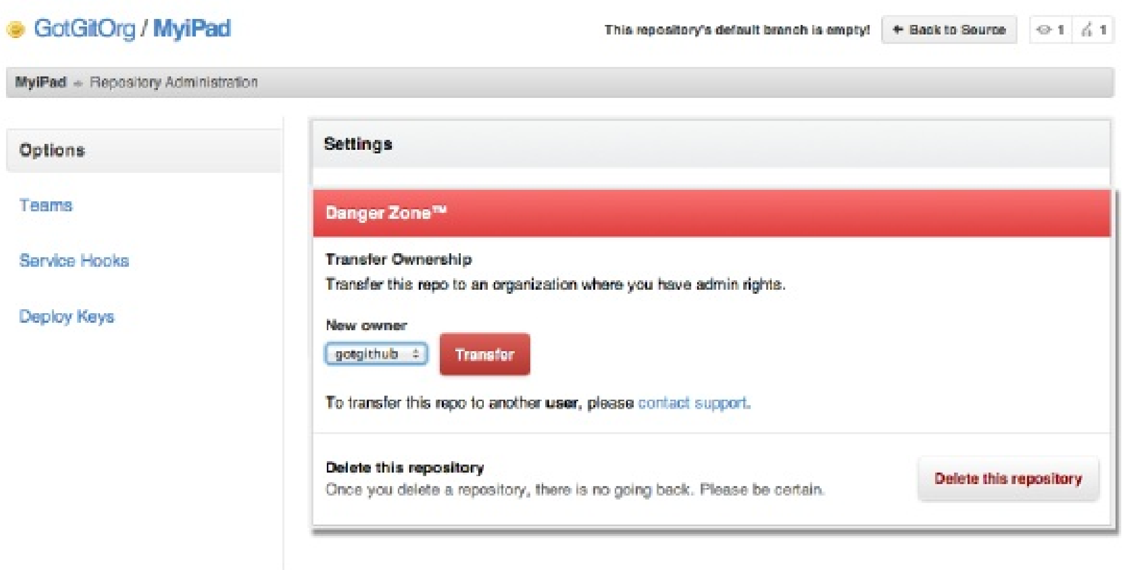


图4-36：项目转移

4.3.4. 个人还是组织

若使用“Fork + Pull”的工作模式，通过个人账号还是组织账号托管版本库，几乎没有什么差别。如果一定要找出点不同，那就是在向托管版本库提交Pull Request时，邮件通知的用户范围有所不同。

- 对于个人账号，对其托管空间内的版本库发出Pull Request，通知邮件会发送给该个人账号及该版本库设置的所有协作者（如果有的话）的邮箱。
- 对于组织，对其托管空间内的版本库发出Pull Request，不会向组织的邮箱发送Pull Request，也不会向组织的所有者（Owner团队）发送通知邮件，而是向在版本库中拥有Push权限的团队（非Owner团队）成员发送通知邮件。

因此，如果在组织的托管空间创建版本库，一定要要为版本库指派一个拥有Push权限的团队，以免以“Fork + Pull”模式工作时，Pull Request没有人响应。

若是以共享版本库方式（即集中式协同模式）工作的话，使用组织来托管版本库会比使用个人账号托管有效率得多。

- 以个人账号托管，需要逐一为版本库设置协作者（Collaborators），如果版本库较多且授权相同，配置过程繁琐且易出错。
- 以组织方式托管，将用户分组，划分为一个一个的团队（Team），以团队为单位授权则方便得多。
- 如果是以付费账号创建的私密版本库，使用组织方式管理，会有包括只读、读写等更丰富的授权类型，更符合项目管理的实际。

4.4. 代码评注

针对项目的每一次Pull Request就相当于一次代码评审，评审以讨论的形式显示在Pull Request中。

在Pull Request中还能够看到对应的提交（一个或多个），并可以直接针对提交进行代码评注。对于采用集中式协同的项目，即使较少使用 Pull Request，也同样可以使用代码评注。代码评注会触发通知邮件给项目的开发者。

代码评注有两种形式，一种是针对整个提交的评注，另外一种是对代码进行逐行评注。

4.4.1. 提交评注

查看项目的提交历史，从中选择一个提交，如图4-37所示。

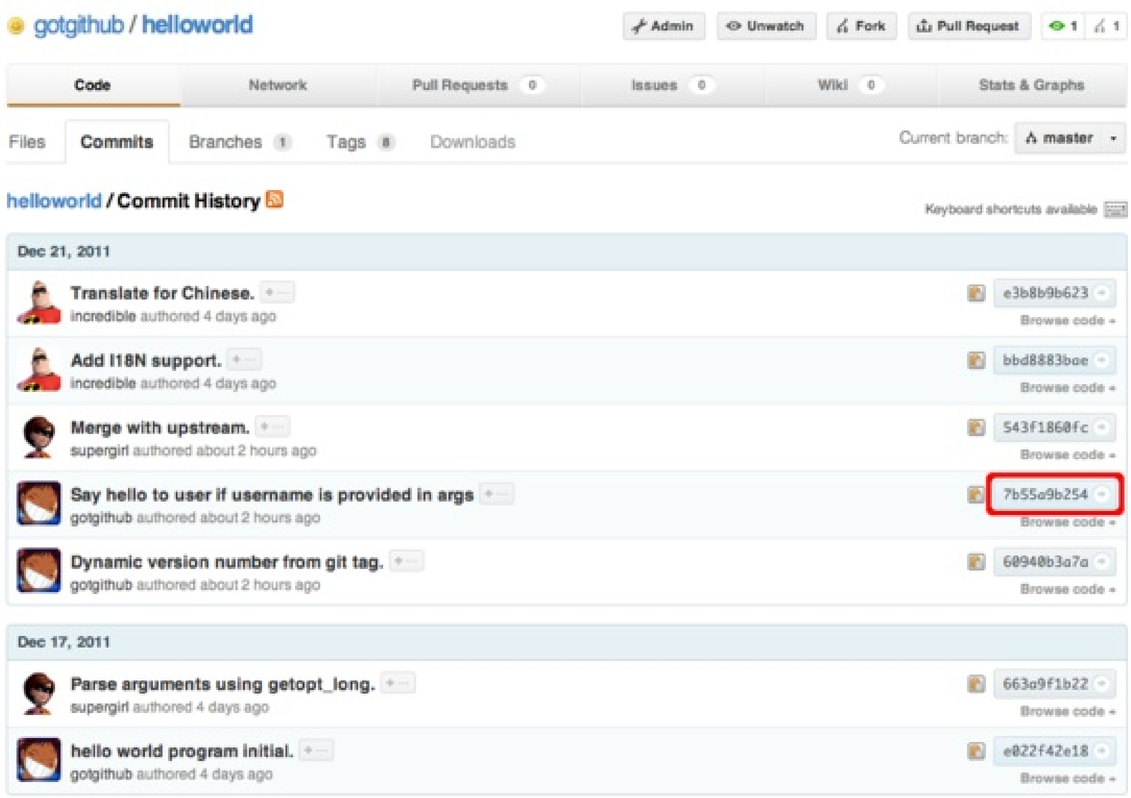


图4-37：helloworld项目提交历史

如图4-38是查看提交的界面。除了提交说明、提交者信息之外，还显示提交所修改的文件和改动差异。在查看提交页面的最下方显示一个提交评注对话框，可以在其中写下评注。评注可以使用 Markdown 语法。

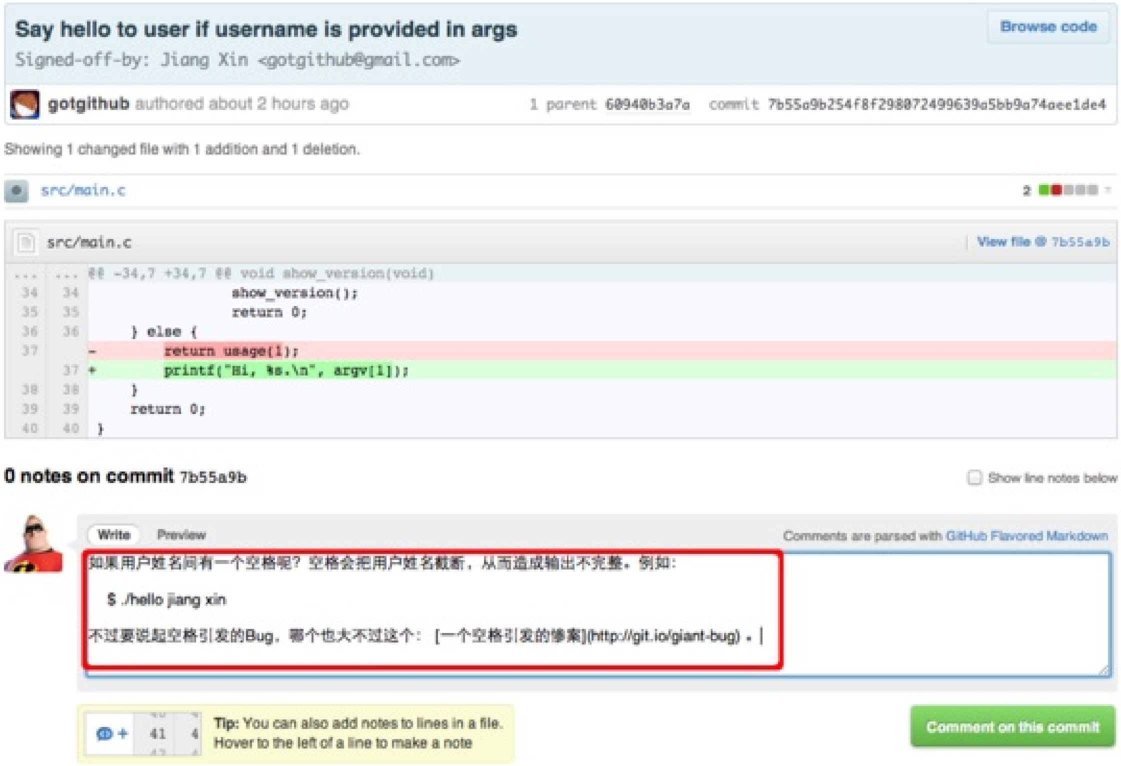


图4-38：添加提交评注

添加评注后，所评注的提交的作者会收到通知邮件，提醒针对自己的提交有了新的评论。通知邮件如图4-39所示。



图4-39：提交评注的通知邮件

通过Web界面可以看到添加在提交下方的评注，并可以撰写新的评注展开讨论。评注者本人或提交的作者还可以编辑甚至删除评注。如图4-40所示。

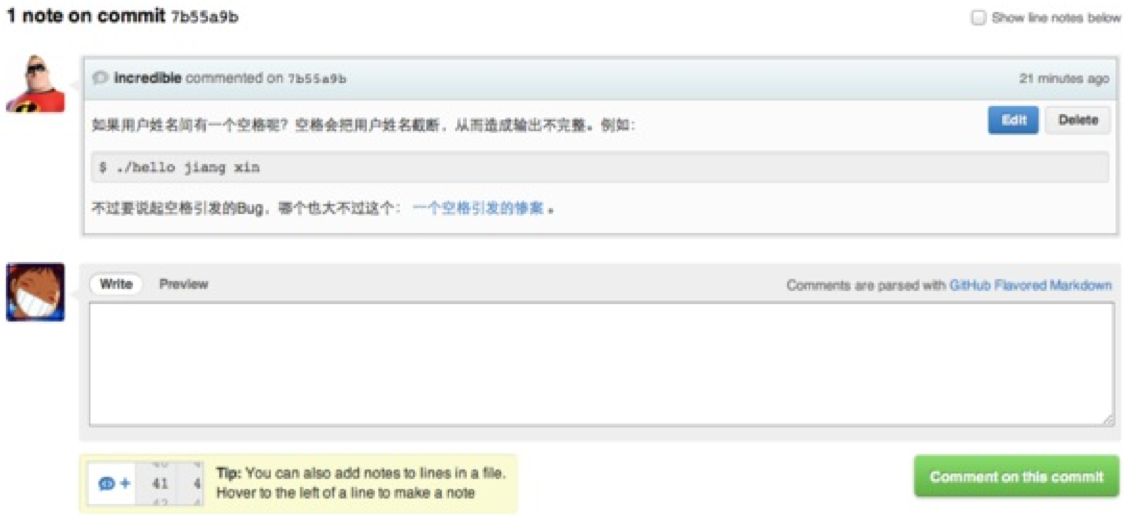


图4-40：提交评注

GitHub还支持Git自身提供的评注功能[1]，如图4-41所示的是提交http://git.io/git-notes(即网址https://github.com/ossxp-com/gitdemo-commit-tree/commit/e80aa74)的评注，这个评注并非通过GitHub添加的，而是由git-note命令提交的评注。这种评注针对一个特定提交只能有一个，GitHub只能显示不能编辑和删除。关于如何通过命令行查看git-note格式的评注，参见《Git权威指南》第570页“41.5 Git评注”。

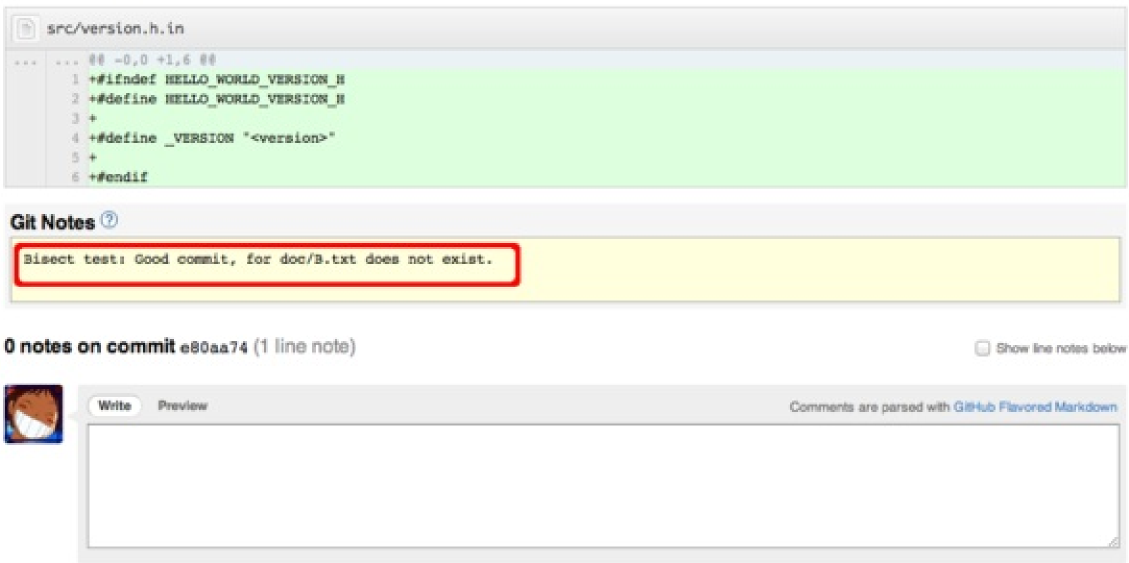


图4-41：git-note评注

4.4.2. 逐行评注

还是以gotgithub/helloworld版本库中的提交为例，看一下GitHub支持的逐行评注功能，即针对提交中的任意一行添加评注。浏览提交，如图4-42所示，当鼠标置于任意一行代码时，在该行代码的左侧会显示一个添加注释的图标。



图4-42：添加逐行评注按钮

点击该图标（用于添加逐行评注的图标），会显示如图4-43所示的添加逐行评注对话框。该评注对话框出现在两行代码之间，在其中写下评注。

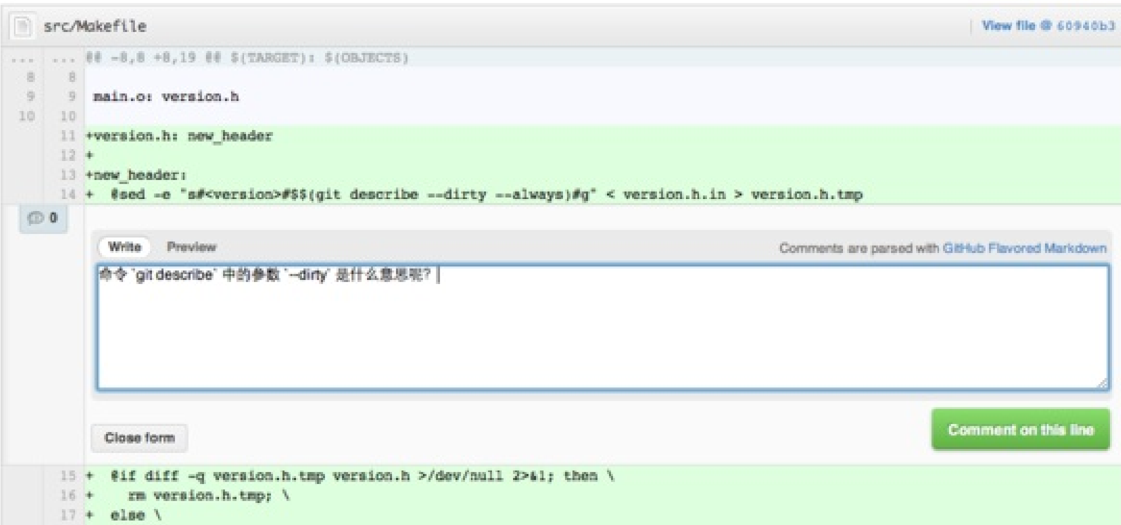


图4-43：添加逐行评注

添加评注后，项目的开发人员同样会收到邮件通知。针对同一行代码的多次评论按时间顺序依次显示，图4-44展示了多个行间评注，其中一个评注还使用 Markdown 语法嵌入了一个图片。

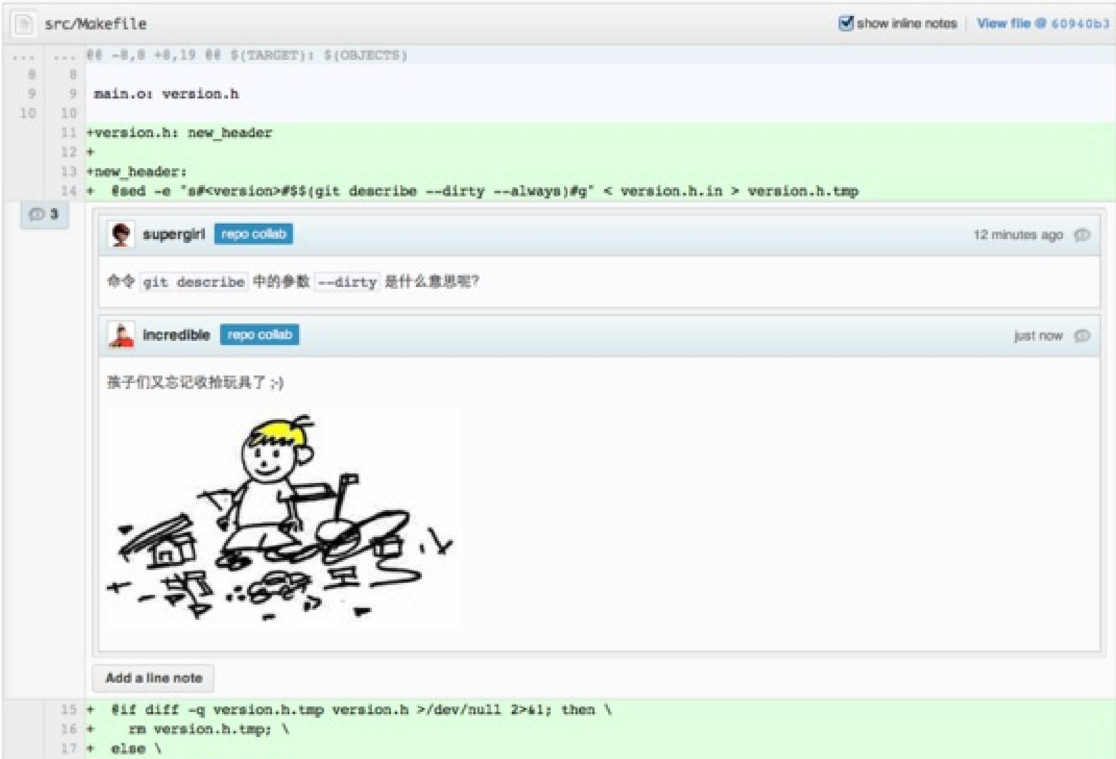


图4-44：逐行评注和提交评注

更有意思的评注可以围观MrMEEE/bumblebee项目的一个bug修正提交（被戏称一个空格引发的惨案）。地址： http://git.io/giant-bug(即网址 https://github.com/MrMEEE/bumblebee/commit/a047be85247755cdbe0acce6)。

4.5. 缺陷跟踪

缺陷跟踪（Bug Tracking）是软件研发流程中重要的一环，集项目需求管理和缺陷管理于一身，通过对研发工作流的控制帮助团队建立规范的研发体系。GitHub提供轻量级的缺陷跟踪模块，称为Issues。小巧、易用的Issues模块能与Pull Request紧密整合，是Pull Request工作流的有益补充。

一个小型、管理文档和网页的项目，使用Pull Request往往就足够了。试想如果贡献者能够直接修改代码（Fork and edit this file）并通过Pull Request贡献给项目核心开发者，那么为什么还要通过Issues模块报告错误并由他人来更改呢？但是对于大型项目需要做需求管理，或者参与代码开发有难度，则非常有必要通过Issues模块启用缺陷跟踪系统，提供更多途径让贡献者参与到项目中来。

缺陷跟踪可以通过项目的管理页面开启或关闭，如图4-45所示。

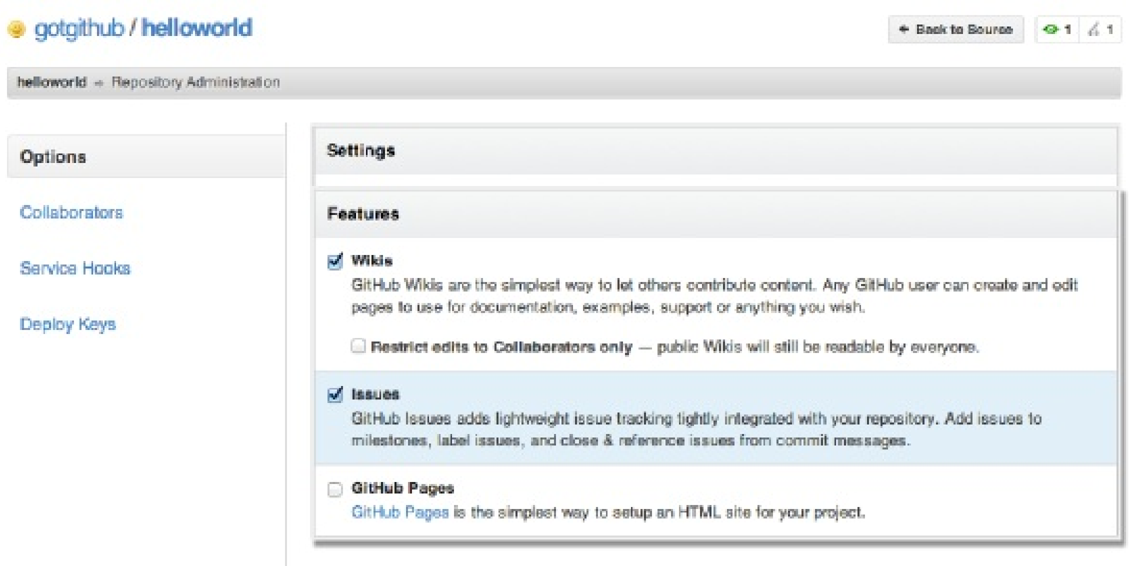


图4-45：开启或关闭Issues模块

4.5.1. 标签

缺陷跟踪系统通常可用于管理多种不同类型的问题：需求、缺陷或其它，也可以通过项目不同模块、组件来为问题分类。GitHub在问题分类的实现上非常简单，通过标签（label）来为问题建立分类。

开启Issues模块后，项目的菜单中多出一个“Issues”项，点击则进入问题浏览界面，如图4-46所示。左侧的边栏是问题过滤器，由上至下分为三个部分。最上面的过滤器根据问题的所有者对问题进行筛选，默认选择所有人的问题。中间的过滤器是根据里程碑对问题进行筛选，默认未选定任何里程碑（初始尚未创建任何里程碑），不对问题进行里程碑过滤。最下面的过滤器依据问题标签对问题进行筛选，初始标签尚未创建。

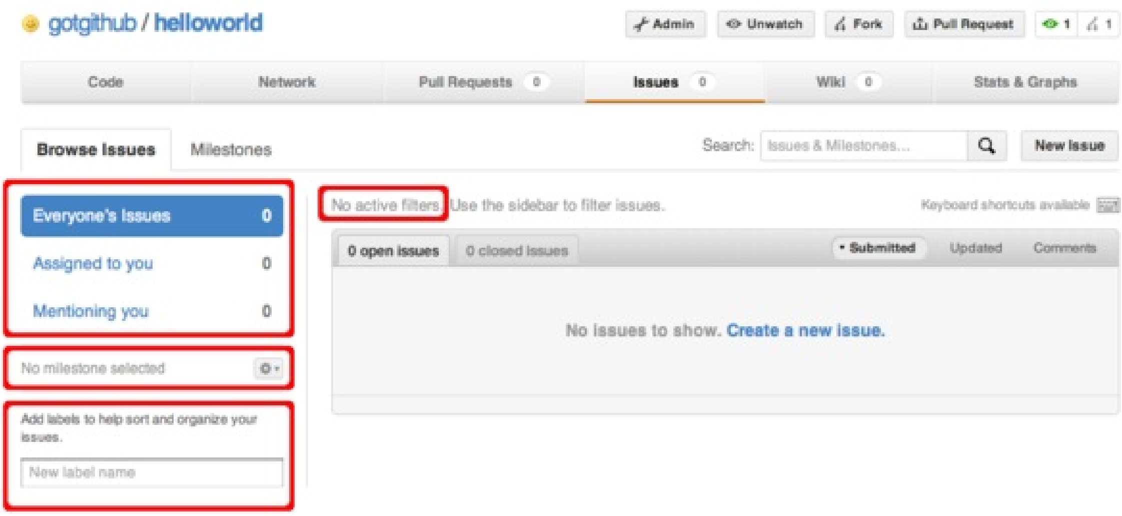


图4-46：尚未定义标签

鼠标点击左侧边栏标签过滤器中的新建标签的文本框，显示如图4-47所示的新建标签界面。输入新的标签名，并为标签选择一个颜色，创建新的标签。

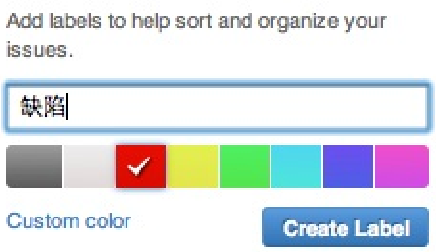


图4-47：创建新标签

创建的标签显示在过滤器中，如图4-48所示。点击过滤器中的标签则对问题进行筛选，取消过滤器可以点击图中标记的“Clear active milestone and label filters”链接。

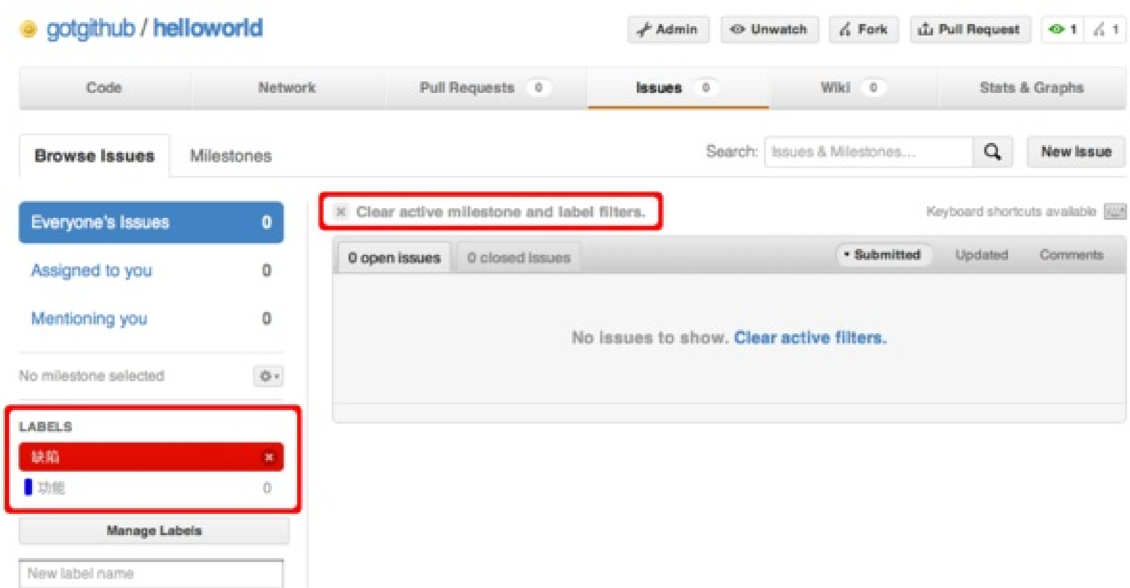


图4-48：过滤器中的标签选项

标签过滤器下方的“Manage Labels”按钮用于管理标签，下面的输入框用于创建新的标签。

4.5.2. 里程碑

里程碑（Milestones）是项目进度管理的重要工具。在传统项目管理中，里程碑对应于一个项目开发计划、一个软件版本；在敏捷项目管理中，里程碑对应于一个Sprint（冲刺）；在软件代码的版本库中则对应于一个标签（tag）或分支（branch）。

在Issues模块中的“Milestones”页面用于里程碑管理。创建新的里程碑需要输入里程碑名称和里程碑的截止时间，如图4-49所示。

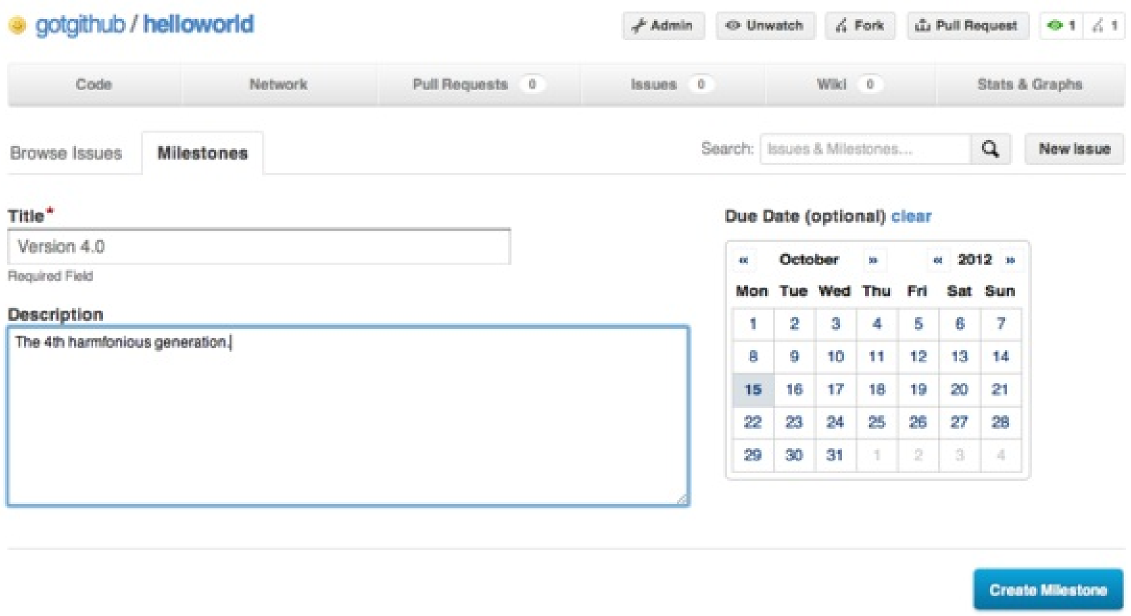


图4-49：创建新里程碑

创建的里程碑以进度条形式显示在里程碑页面中，如图4-50所示定义了两个里程碑。这两个里程碑的时间跨度定义的太长，敏捷的项目管理从来不这么定义。

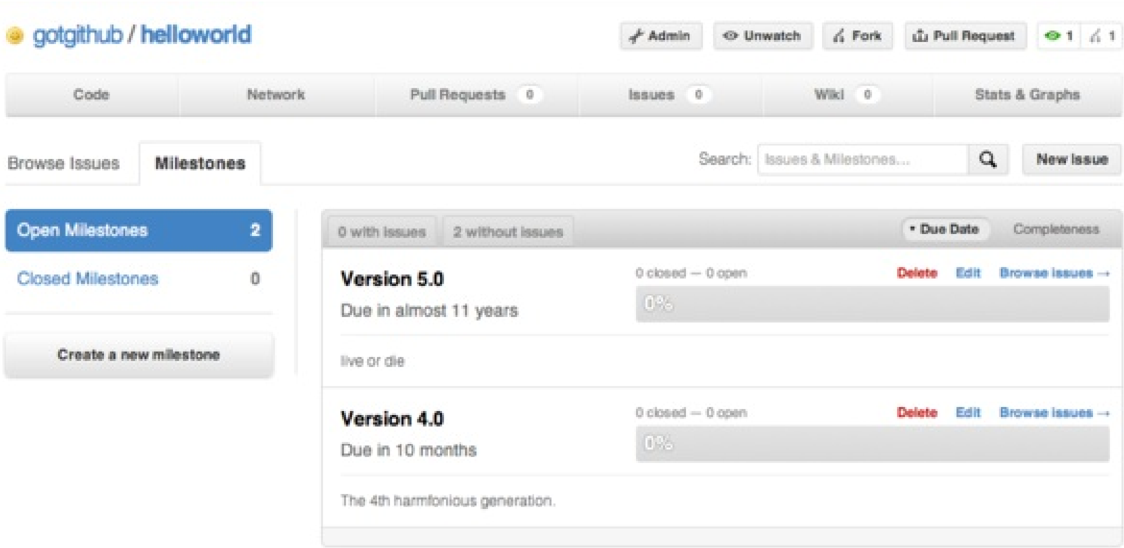


图4-50：里程碑列表

4.5.3. Issue的生命周期

GitHub的Issues模块非常简单，对标签和里程碑进行简单的设置后，基本上就完成了Issues模块的配置工作，接下来就是如何创建和修改Issue，完成项目的缺陷跟踪和需求管理等，这才是Issues模块的主要工作。

每个Issue都有自己的生命周期，从问题的创建，到问题的指派，再到问题的解决，直至问题的关闭。图4-51就是以普通贡献者身份为项目创建Issue。

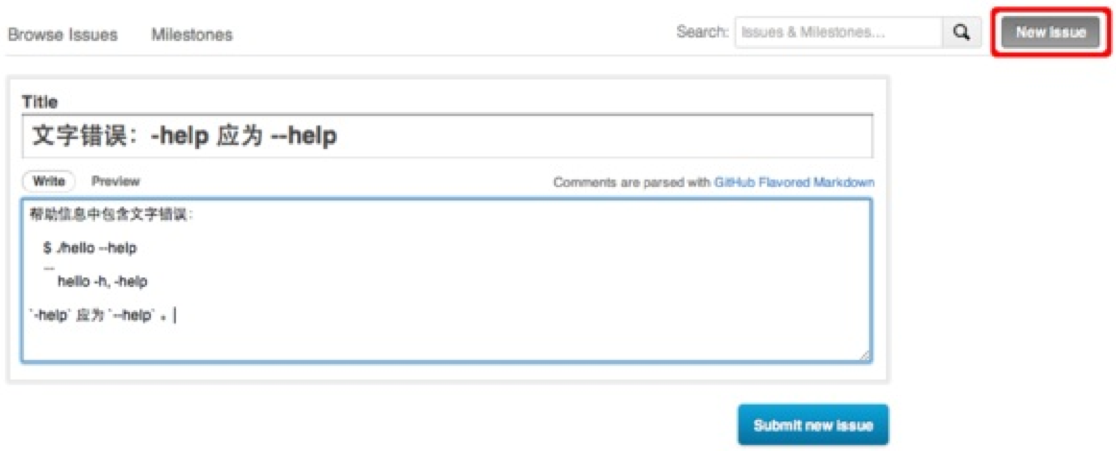


图4-51：以普通贡献者身份创建问题

录入问题标题和描述后，点击“Submit new issue”按钮，完成问题创建。图4-52显示了新建立的问题，可以看出新建问题尚未设置标签。

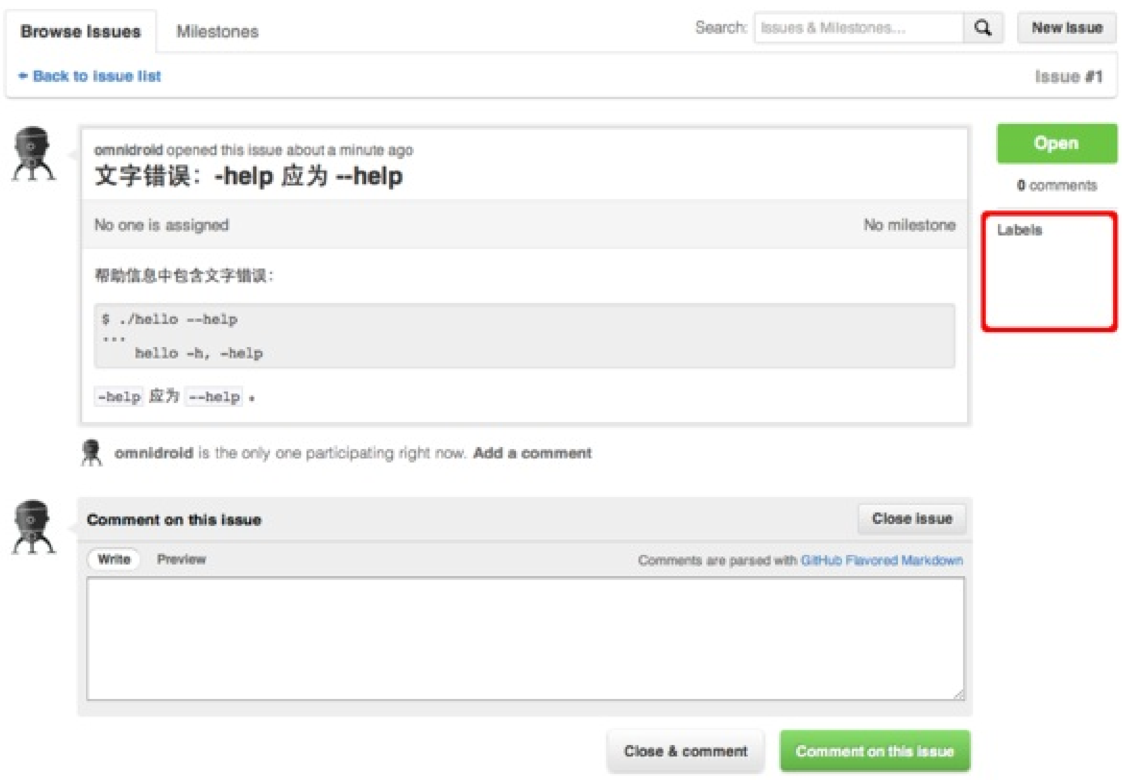


图4-52：新创建的问题尚未添加标签等

普通贡献者创建问题时只能录入问题的标题和描述，而不能设置问题的指派（谁来负责）、添加标签和设置里程碑。如果希望问题通知到特定的开发者，可以在问题描述中以“@用户名”的方式通知到该用户[1]，这也是众多社交软件通行的做法。

项目成员创建问题时，拥有更大权限，也有更多的可选项。如图4-53所示。

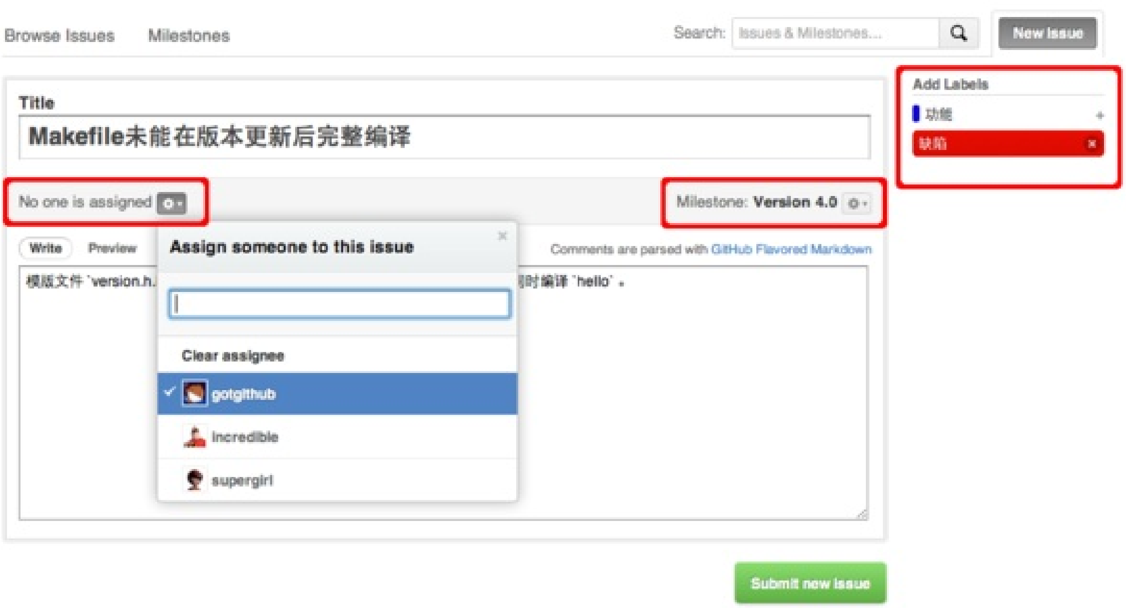


图4-53：以项目成员身份创建问题

完成上述两个问题的创建后，问题浏览界面显示新创建的两个问题，一个以项目成员身份创建的问题已经被设置了“缺陷”的标签，而另外一个问题则没有设置任何标签。如图4-54所示。

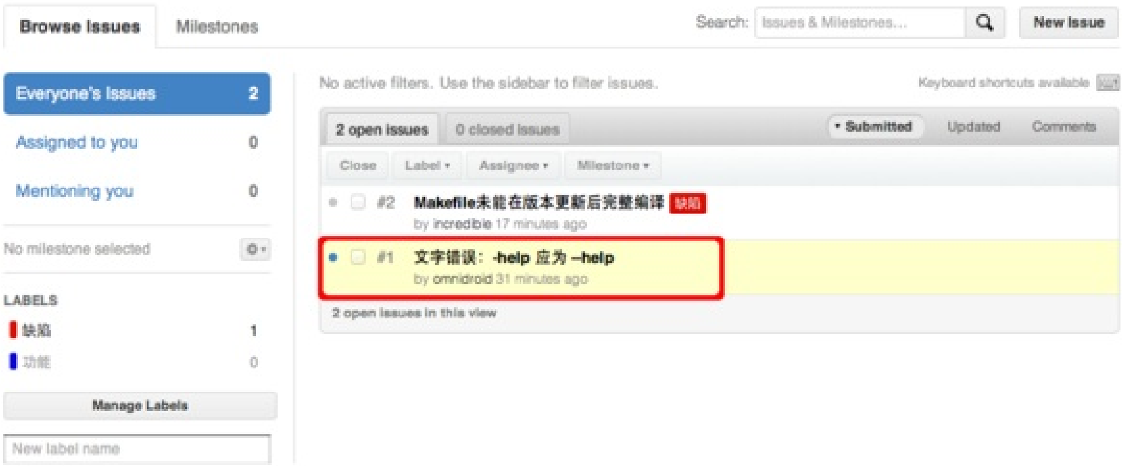


图4-54：所有问题列表

以项目成员身份登录，在问题浏览界面即可为问题重新设定标签，指派负责人，设置里程碑，以及关闭问题等。如图4-55所示。

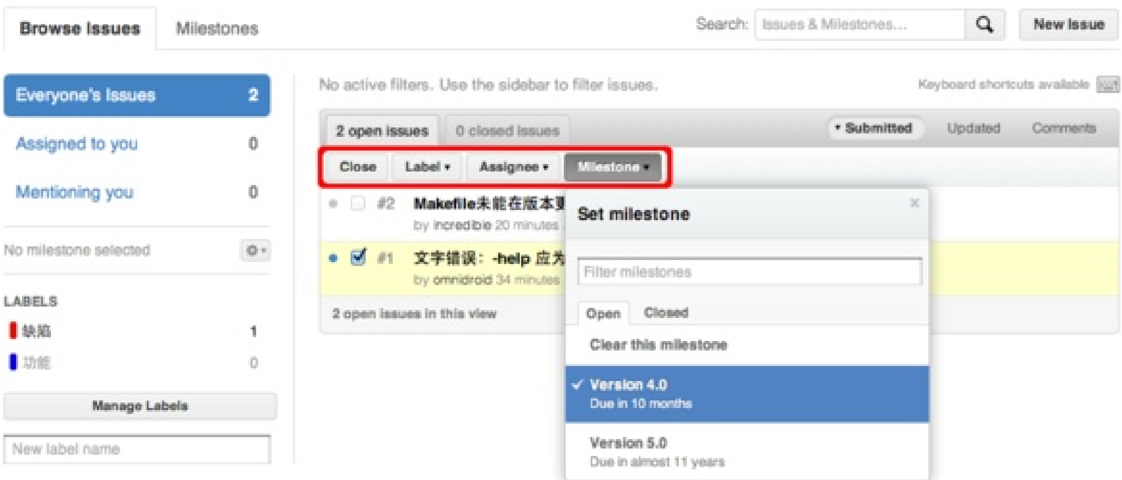


图4-55：为问题添加指派、里程碑和标签

在问题浏览页面的过滤器中选择里程碑”Version 4.0“，可以看到两条问题都显示出来，这是因为这两条问题都属于该里程碑。里程碑的进度条显示进度为零，这是因为里程碑所包含的全部（两个）问题都处于打开状态，尚未解决。如图4-56所示。



图4-56：通过里程碑筛选问题

邮件通知功能是缺陷跟踪系统推动工作流的重要工具，GitHub的Issues模块也具有邮件通知功能。除了像其他缺陷跟踪系统在收到邮件通知后，访问Web界面参与问题的讨论外，还可以直接以邮件回复的功能参与到工作流中[2]。

GitHub还支持版本库提交和问题建立关联，只要提交说明中出现“#xxx”（Issue编号）字样。如果在提交说明中的问题编号前出现特定关键字，还可以关闭问题。支持的关键字有：

- fixes #xxx
- fixed #xxx
- fix #xxx
- closes #xxx
- close #xxx
- closed #xxx

下面就以gotgithub/helloworld版本库为例，关闭编号为“#1”的问题。

- 克隆版本库，若本地工作区尚不存在。

```
$ git clone git@github.com:gotgithub/helloworld.git
$ cd helloworld
```

- 编辑文件src/main.c，改正“问题#1”发现的文字错误。

```
$ vi src/main.c
$ git diff
diff --git a/src/main.c b/src/main.c
index 3daf9fe..f974b49 100644
--- a/src/main.c
+++ b/src/main.c
@@ -19,7 +19,7 @@ int usage(int code)
            "            say hello to the world.\n\n"
            "    hello -v, --version\n"
            "            show version.\n\n"
-           "    hello -h, -help\n"
+           "    hello -h, --help\n"
            "            this help screen.\n\n"), _VERSION);
     return code;
 }
```

- 将修改添加至暂存区。

```
$ git add -u
```

- 提交，并在提交说明中用fixed #xxx关键字关闭相关问题。

```
$ git commit -m "Fixed #1: -help should be --help."
```

- 向GitHub版本库推送。

```
$ git push
```

推送完毕后，在问题浏览界面可以看到里程碑“Version 4.0”的进度已经完成了一半，即其中一个问题（#1）已经完成并关闭。如图4-57所示。

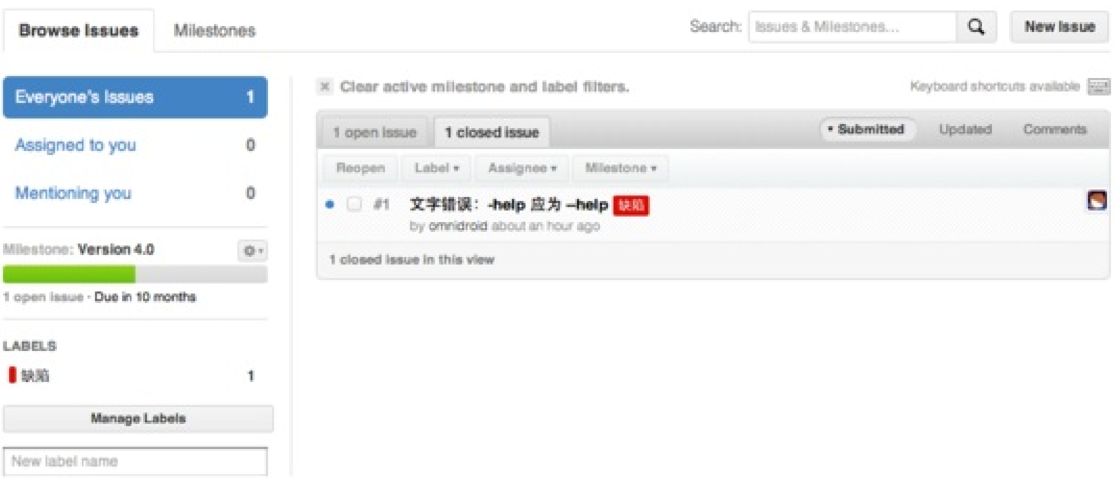


图4-57：关闭一个问题，里程碑完成50%

查看已经完成的问题（#1），可以看到其中关联到一个提交，该提交正是我们刚刚创建的。如图4-58所示。

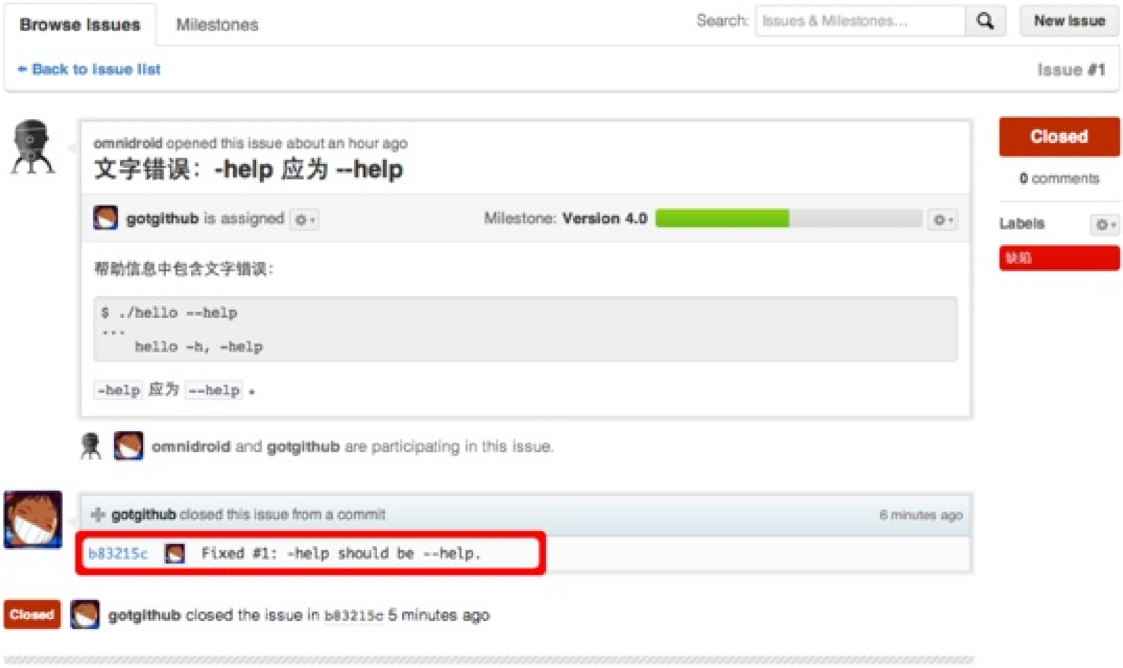


图4-58：已关闭问题中的提交链接

点击关联的提交，显示如图4-59的提交界面，出现在提交说明中的问题编号也可点击，指向缺陷追踪系统中该问题的链接。

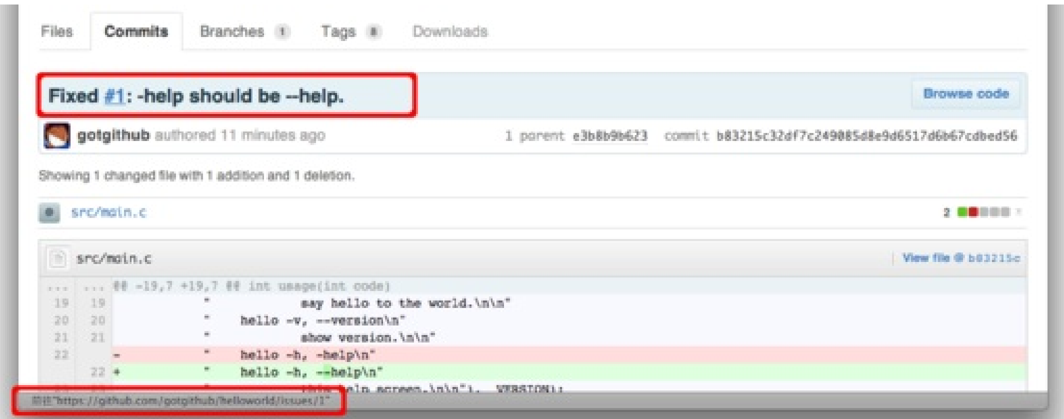


图4-59：提交中的问题链接

4.5.4. Pull Request也是Issue

Pull Request和Issue一样，也是一种对项目的反馈，而且是更为主动的反馈。GitHub的Issues模块将Pull Request也纳入到问题的管理之中，完美地将Pull Request整合到问题追踪的框架之中。

为了弄清二者之间的关联，首先创建一个Pull Request。

以非项目成员（如用户 omnidroid）的账号访问gotgithub/helloworld项目，查看文件src/Makefile，点击“Fork and edit this file”按钮快速创建派生项目，如图4-60所示。

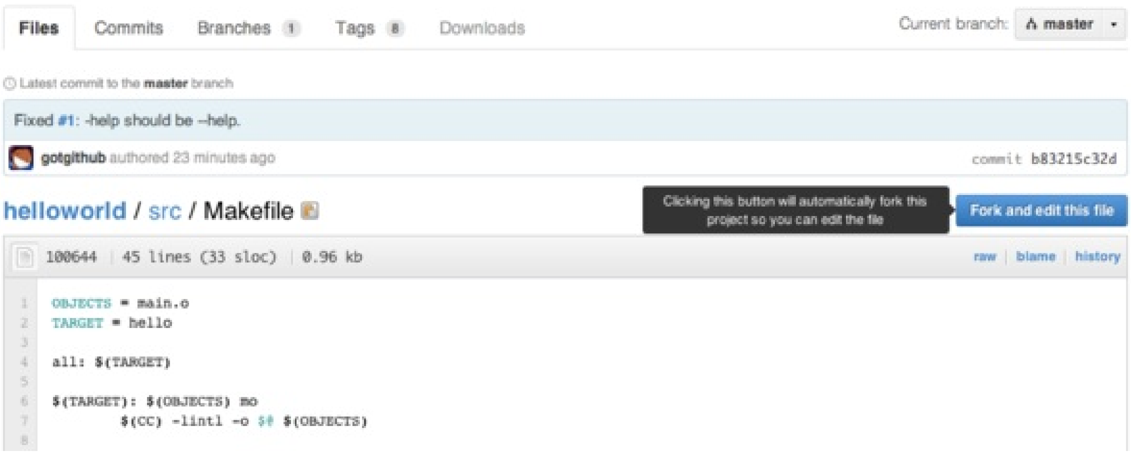


图4-60：在线编辑并创建派生项目

通过GitHub提供的在线编辑功能修改src/Makefile文件，修改完毕后撰写提交说明，点击“Propose File Change”按钮提交。如图4-61所示。

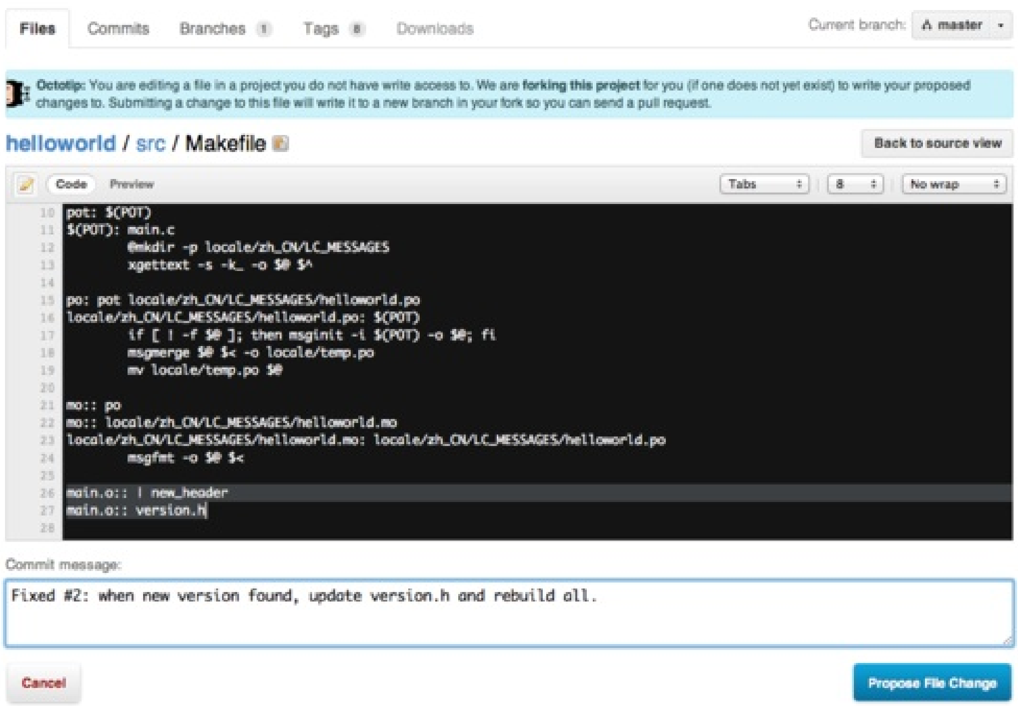


图4-61：在线编辑并提交

在提交说明中特意使用了“Fixed #2”关键字，以便该提交被上游版本库接纳后能够关闭关联的问题。

当完成提交后，GitHub会自动开启创建新的Pull Request对话框，如图4-62所示。

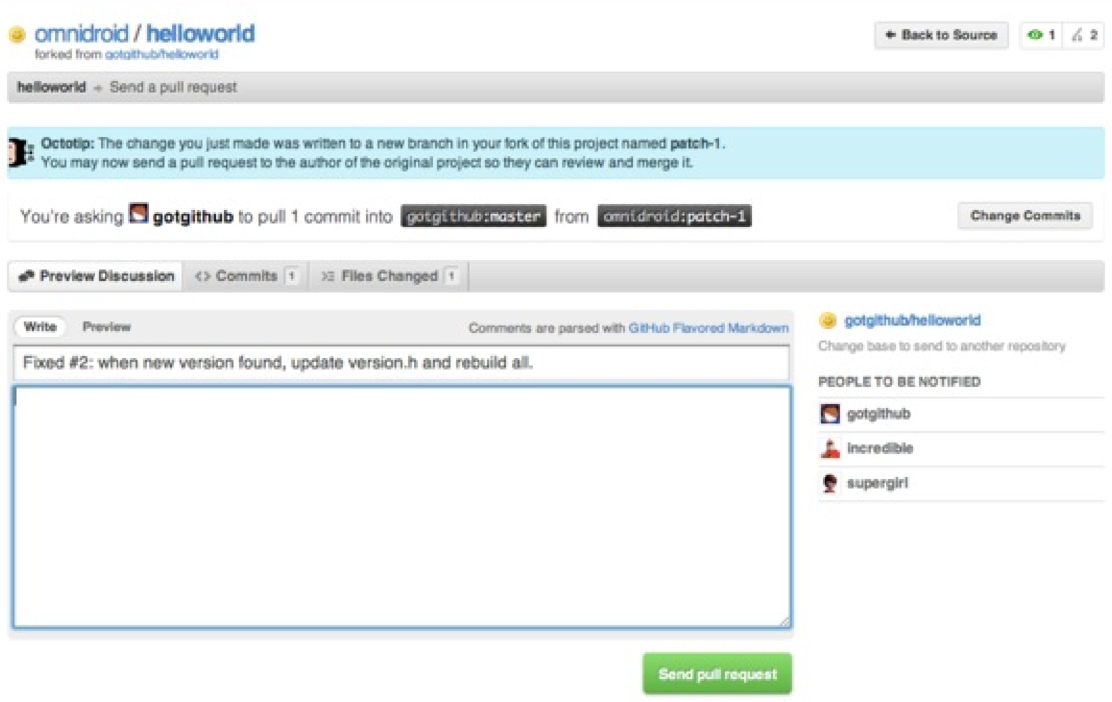


图4-62：创建Pull Request

Pull Request创建完毕后，除了在菜单项“Pull Requests”中有显示外，在“Issues”的问题浏览页面中也会显示。如图4-63所示，新建立的Pull Request的编号不是从壹开始创建，而是接着问题的编号顺序创建，所以当Pull Request出现在问题列表中时，如果不注意后面的山型的分支图标，根本意识不到这不是一个普通的问题（Issue），而是一个Pull Request。

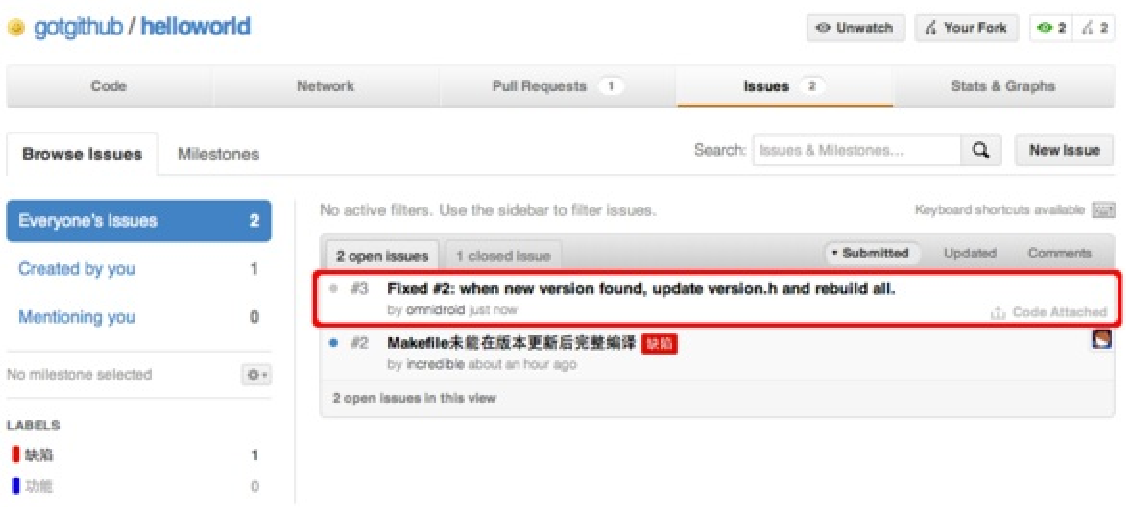


图4-63：Pull Request也显示在Issues中

显示在问题浏览界面中的Pull Request和问题一样，可以为其设置标签、指派负责人、设置里程碑。如图4-64所示。

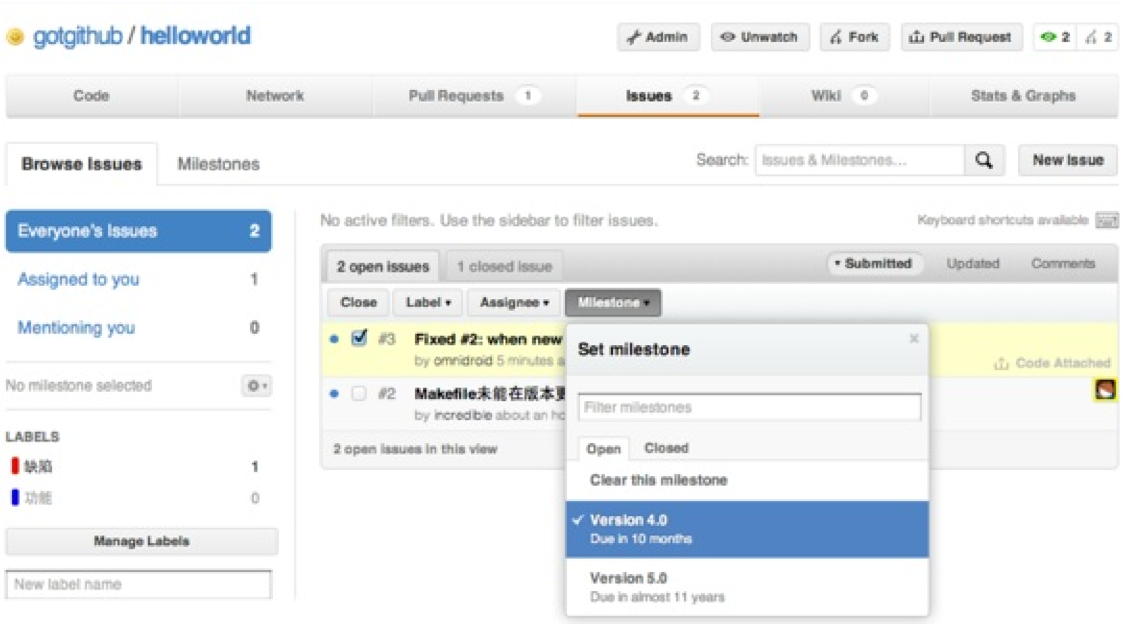


图4-64：可以像更新其他Issue那样更新Pull Request

当Pull Request归类到里程碑“Version 4.0”中时，在过滤器打开里程碑“Version 4.0”，可以看到本来已经完成50%的进度，由于新增了一个“问题”（Pull Request），导致进度降低了。如图4-65所示。

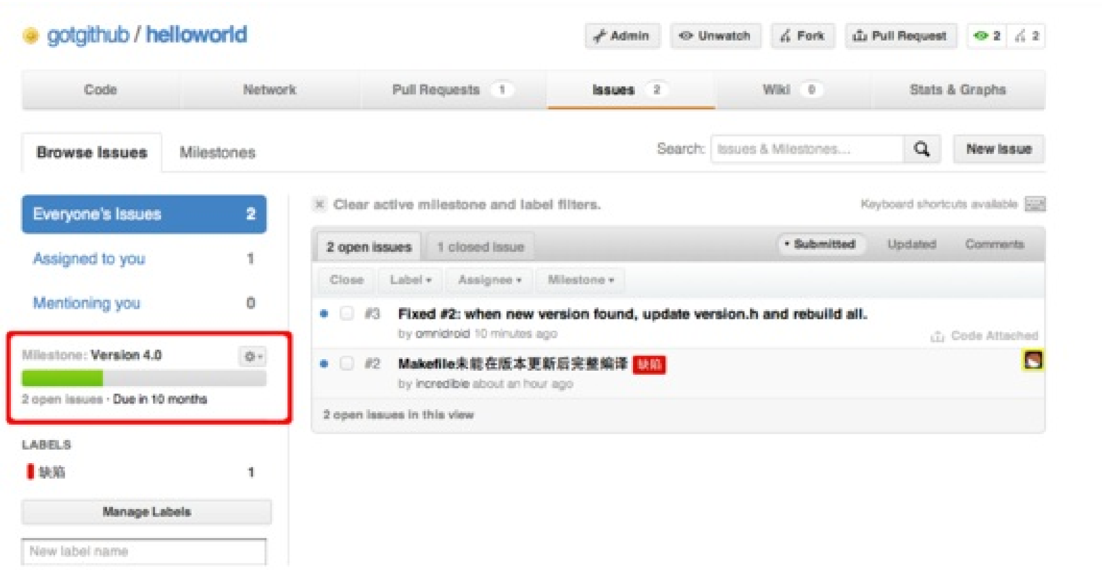


图4-65：里程碑进度调整

点击编号为“#3”的问题（Pull Request），会进入到Pull Request页面。点击页面中的“Merge pull request”按钮实现Pull Request的合并。如图4-66所示。

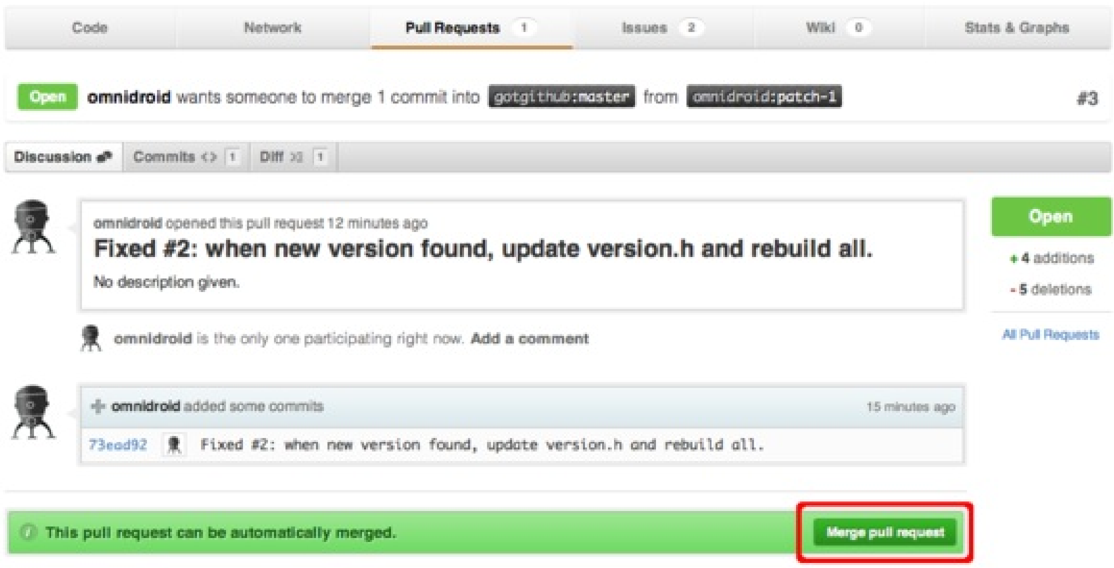


图4-66：在线合并Pull Request

点击“Confirm Merge”确认合并，如图4-67所示。

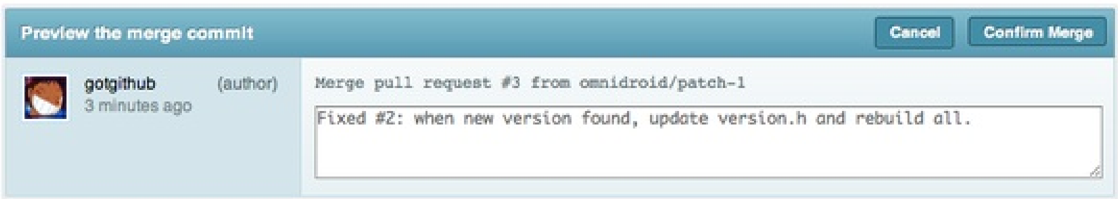


图4-67：确认合并Pull Request

完成合并后，查看该Pull Request，可以看到该Pull Request已经关闭。如图4-68所示。

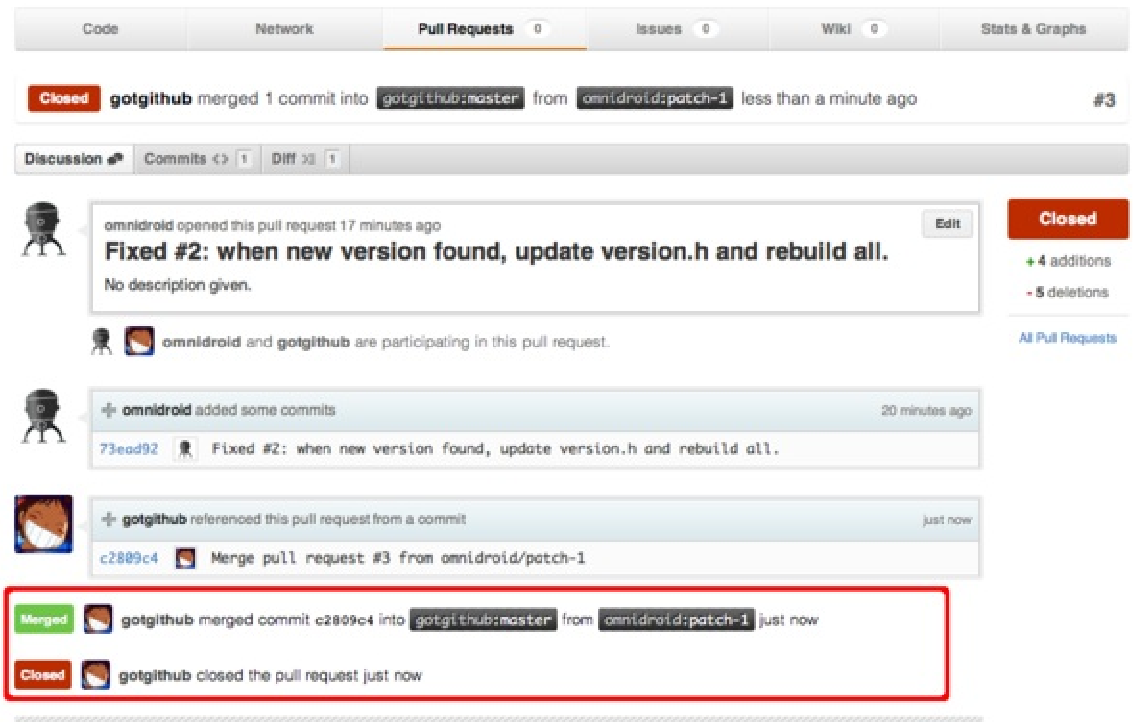


图4-68：Pull Request自动关闭

如果再回到问题浏览界面，能够猜到现在里程碑“Version 4.0”的进度是多少么？

由于关闭了编号为“#3”的Pull Request，以及所合并的Pull Request中对应提交的提交说明的指令同时关闭了编号为“#2”的问题，所以现在里程碑“Version 4.0”关联的所有问题均已关闭。里程碑也显示已关闭，即里程碑完成度为100%。

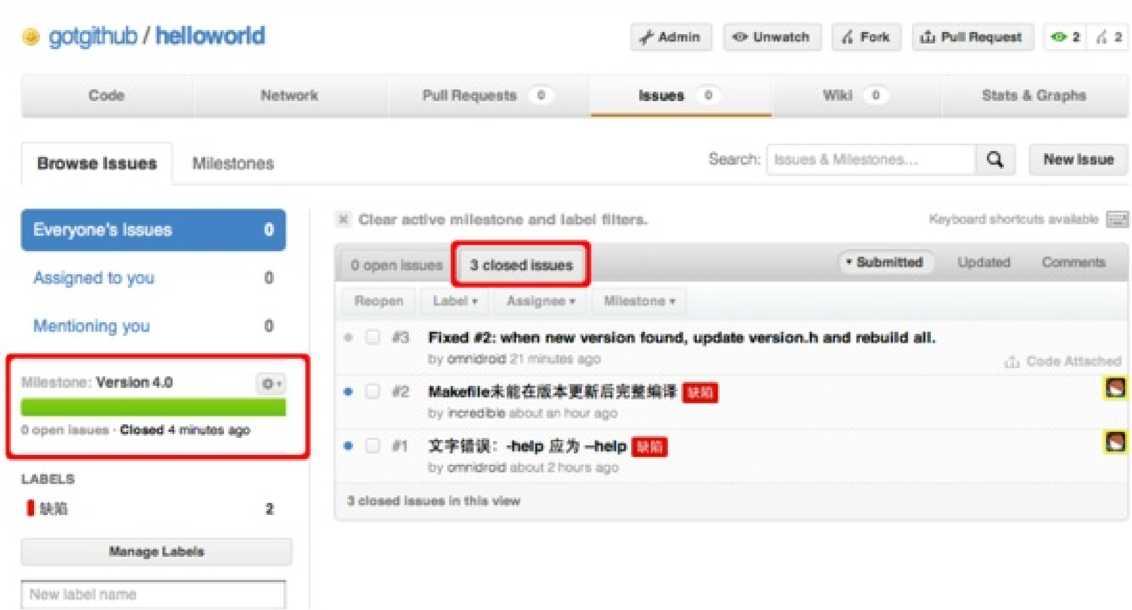


图4-69：里程碑关闭

4.6. 维基

维基是Web协同著作平台，可以让任何浏览网页的人都能够方便地参与网页的编辑和创建。这源自于维基如下魔力：

- 快速更改。修改网页无需复杂的后台修改和网页部署流程，浏览的网页直接提供编辑按钮，任何查看网页的用户均可在线编辑网页。
- 简洁语法。编写网页不需要学习复杂的HTML，取而代之的是易学易用的格式化文本（维基语法），有的维基还提供图形化编辑界面。
- 版本控制。熟悉Git的人，可以把维基看作是Web的版本控制。历次修改都记录在案，历史修订可进行比较，可恢复到历史版本等。
- 维基链接。页面链接使用[[页面名称]]语法，可以非常方便地创建新页面，并实现页面间的互联。

GitHub提供了维基模块，方便项目团队创建社区驱动和维护的项目文档。

4.6.1. 维基初始化

GitHub的维基模块可以通过项目管理页面控制开启或关闭，默认开启，如图4-70所示。因为GitHub提供了项目展示的多种途径，一些小项目如果觉得用README文件构建项目说明，或者用gh-pages分支维护项目主页就足够了，大可关闭维基模块。

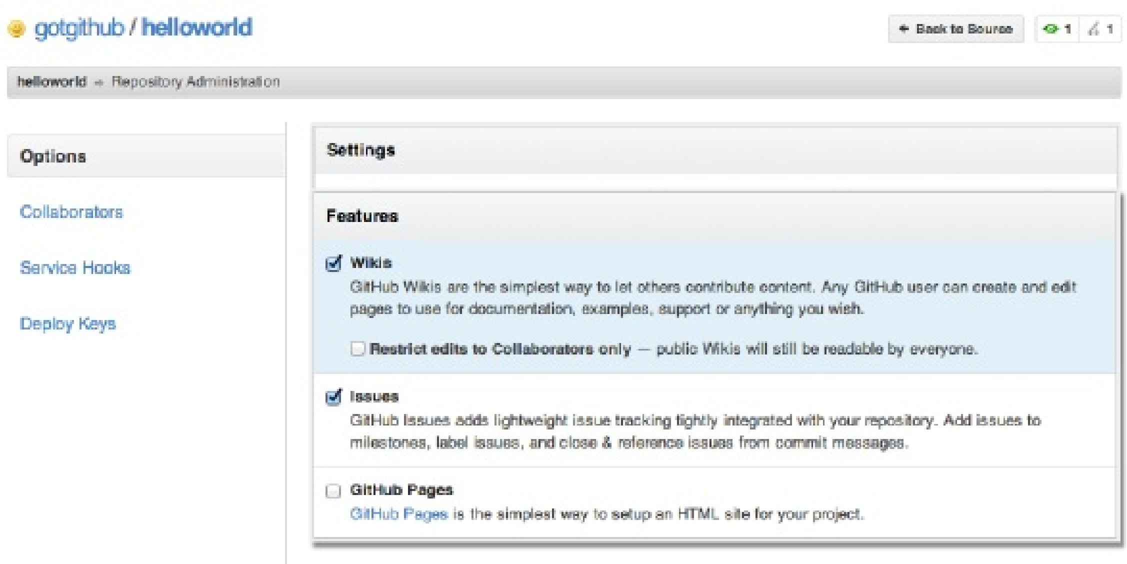


图4-70：开启或关闭Wiki模块

项目启用维基后，进入维基页面，如图4-71所示，会发现维基页面并没有自动创建，还需要进行初始化。

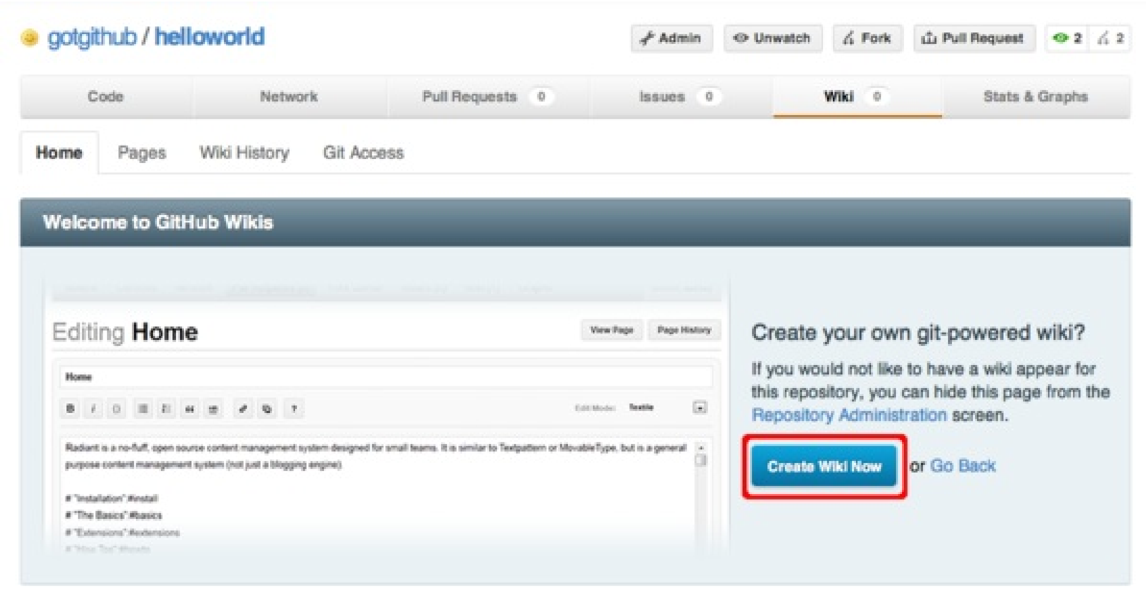


图4-71：尚未初始化的维基界面

点击“Create Wiki Now”按钮，自动创建维基首页，如图4-72所示。

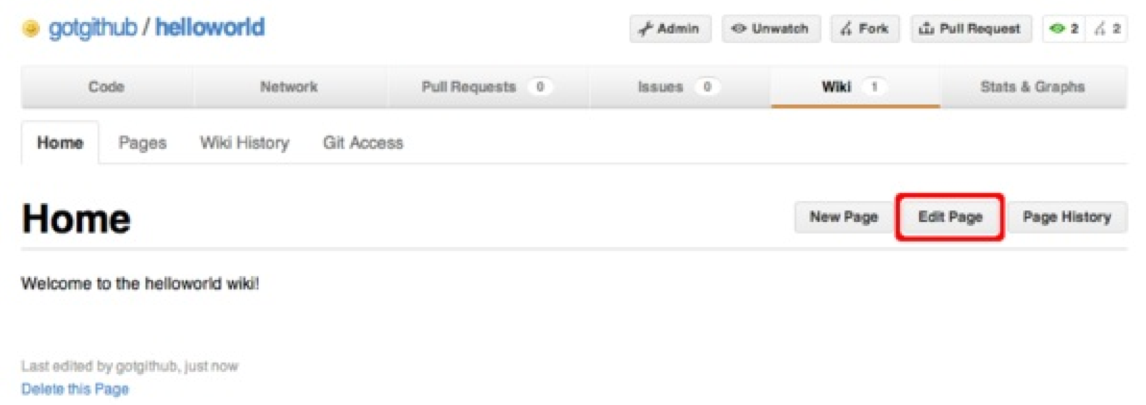


图4-72：自动创建的维基首页

4.6.2. 使用维基

自动创建的维基首页只有非常简单的信息，点击编辑按钮，修改维基首页。编辑界面如图4-73所示。

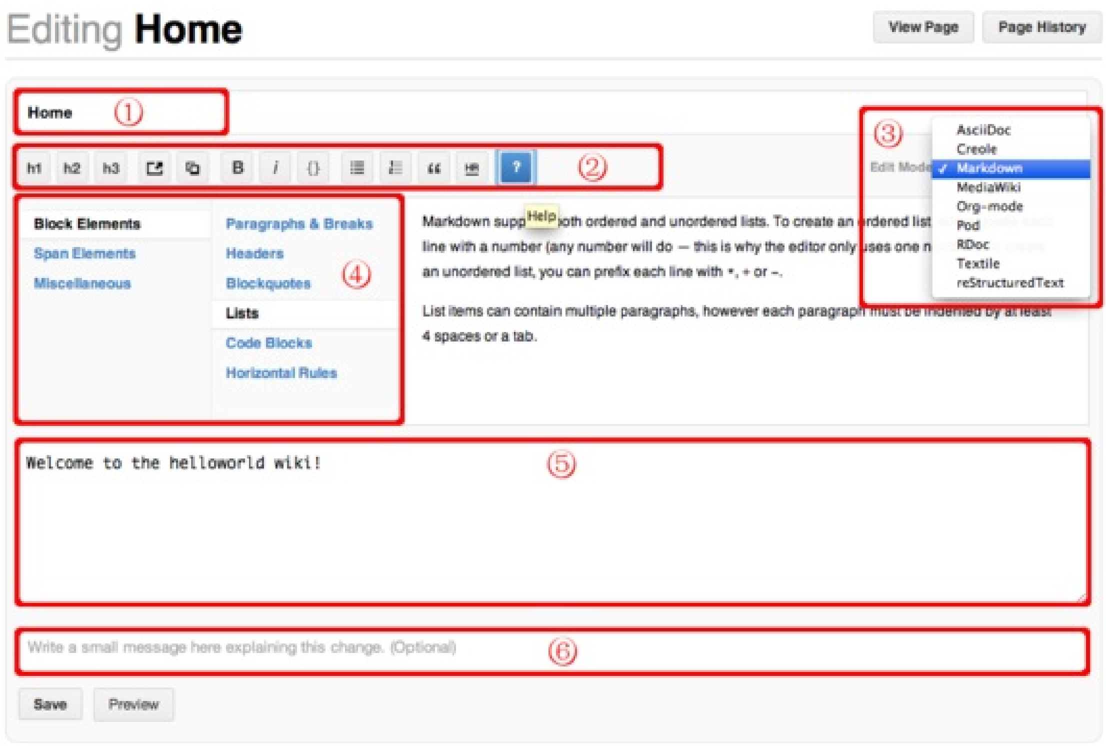


图4-73：编辑维基首页

编辑界面有6个部分需要重点说明：

1. 页面名称。首页的页面名称为“Home”，不能随意更改，否则无法找到首页，或者页面之间的跳转会失效。
2. 工具条。从左至右分别是设置一级标题、二级标题、三级标题、插入链接、图片、字体加粗、斜体、代码块、列表、编号列表、引用、水平分割线等。
3. 页面语法格式。默认采用Markdown语法，还可以选用AsciiDoc、Creole等语法。注意如果改变语法格式，该维基页面的内容需要手工进行调整，而且页面的实际存储文件的文件扩展名会改变。
4. 语法帮助。当按下工具条中的帮助按钮会显示本语法帮助表格。
5. 维基内容编辑框。整个维基页面的内容都在这个编辑框中。鼠标拖动该编辑框右下角可以对编辑框大小进行缩放。
6. 可选的修改说明。修改页面时提供说明便于跟踪对页面的历史修订。

对初始创建的首页进行更改，如图4-74所示。



图4-74：修改选中文本样式

图4-74中，在维基内容编辑框中选择两行文本，然后点击工具栏中的列表按钮，为选中内容应用列表样式。应用新样式后的效果如图4-75所示。然后填写提交说明，点击“Save”按钮保存更改。

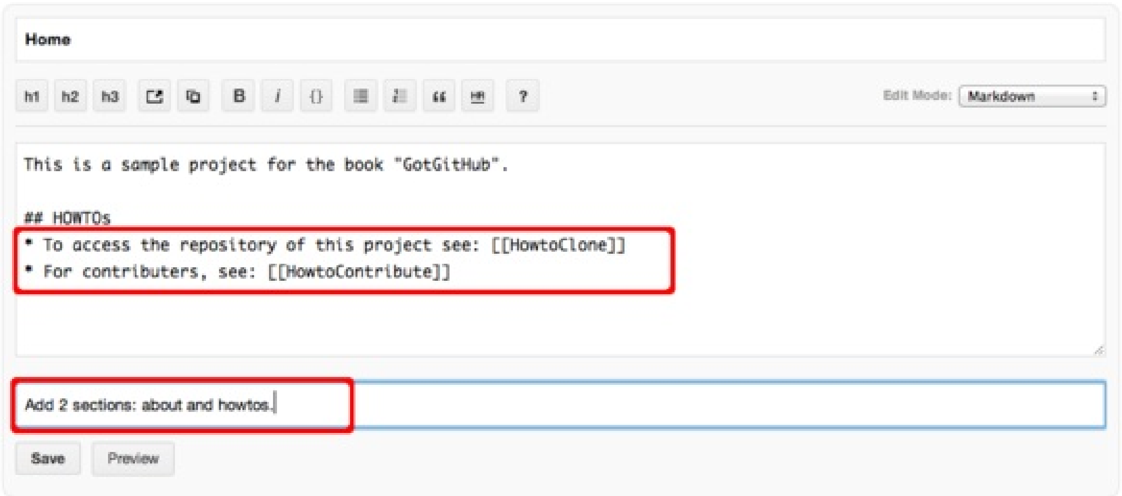


图4-75：填写提交说明保存更改

注意图4-75的维基内容中有[[页面名称]]样式的语法，这个语法是维基特色的页面链接语法，指向另外的维基页面。实际页面输出如图4-76所示。

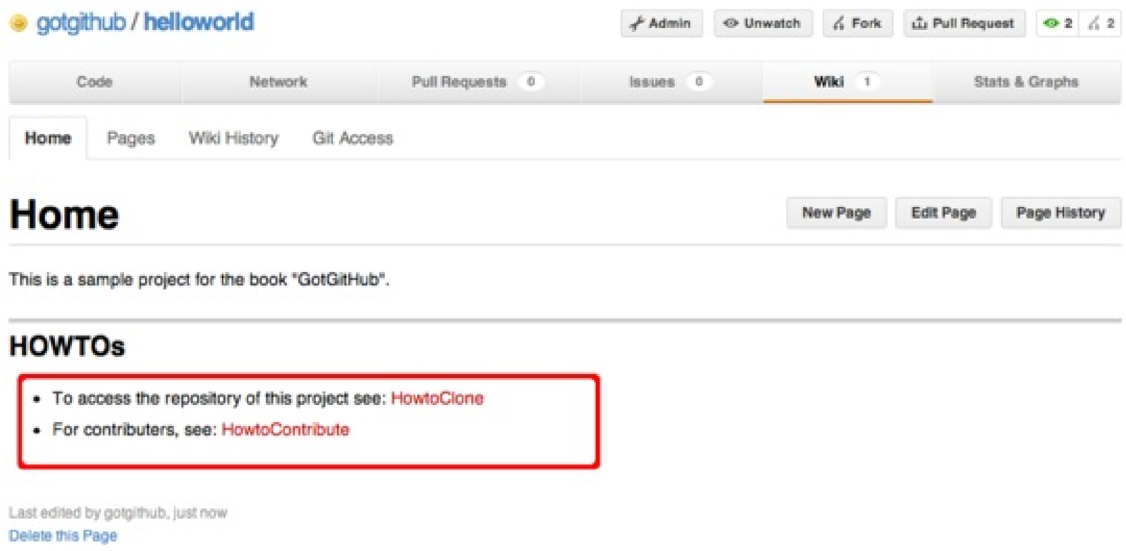


图4-76：保存更改后的维基首页

如果对维基语法[[页面名称]]生成的链接标题不满意，还可以用[[链接标题|页面名称]]格式创建维基链接。对首页重新做一次修改，修改如下：

- 将[[HowtoClone]]改为[[how to clone|HowtoClone]]。
- 将[[HowtoContribute]]改为[[how to contribute|HowtoContribute]]。

修改后的首页效果如图4-77所示。

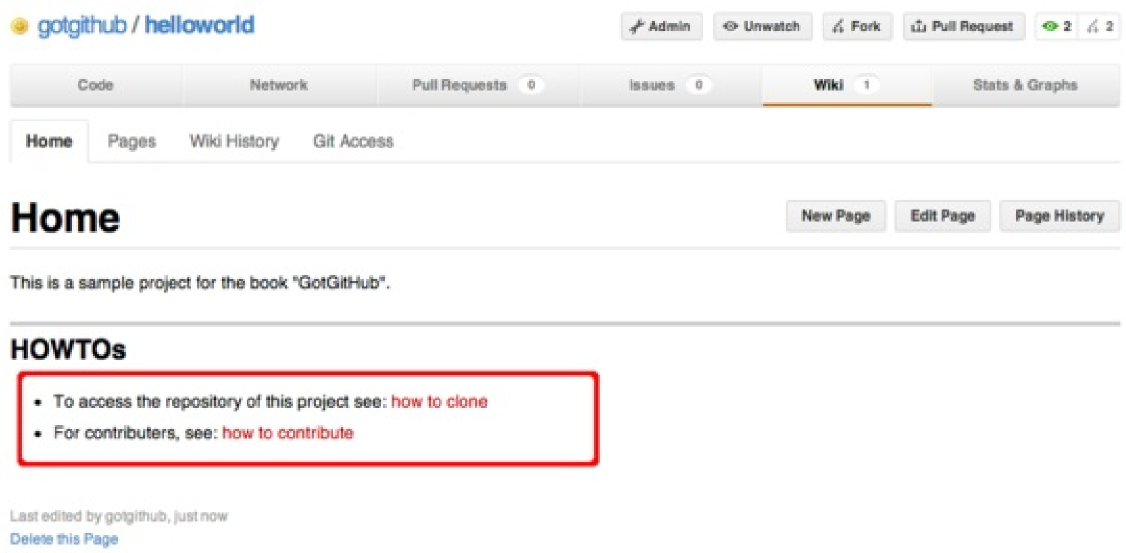


图4-77：修改维基链接标题后的首页

无论怎样更改维基页面都不怕内容丢失，因为维基记录了每一次修订历史，并可以回退任意一次修改。点击维基页面中的“Page History”按钮，查看页面修订历史，如图4-78所示。



图4-78：页面修订历史

首页的修订历史记录着维基初始化以来所有的修改，包括修改者、修改时间、提交说明，以及一个可点击的对象ID。点击对象ID查看对应版本的页面。还可以对不同版本的页面进行比较，选中两个版本点击“Compare Revisions”按钮，如图4-79所示。

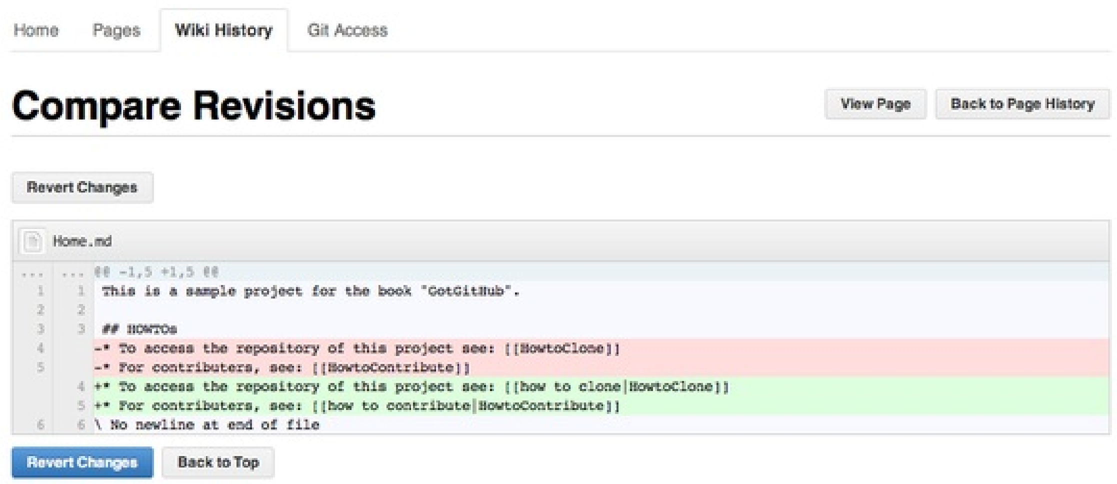


图4-79：页面版本间比较

在页面版本间的比较界面中，提供回退此次修改的按钮。点击“Revert Changes”按钮（图4-79所示），可以回退对首页的修改。查看首页的修订历史，会看到回退记录也显示其中，如图4-80所示。

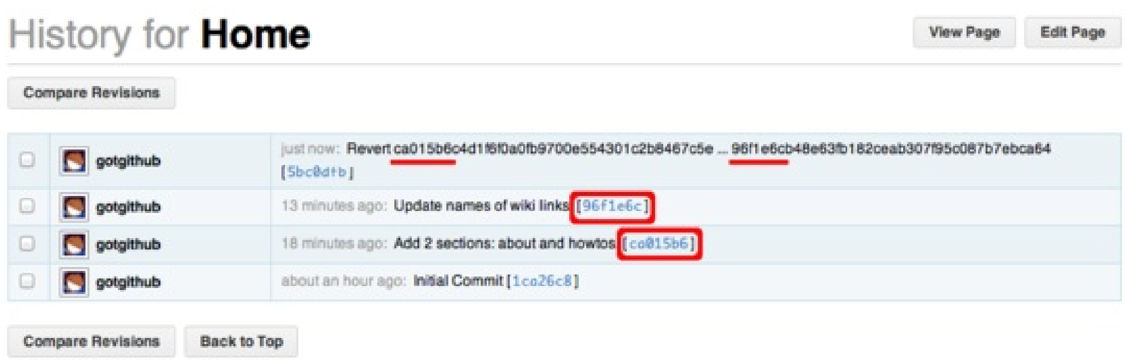


图4-80：包含回退记录的页面修订历史

在维基中创建新页面有多种方法，可以点击页面中的“New Page”按钮，也可以像我们之前做的那样先在页面中用[[页面名称]]格式嵌入维基链接，然后在生成的页面中可以看到指向新页面的链接，当然这些链接所指向的页面并不存在。

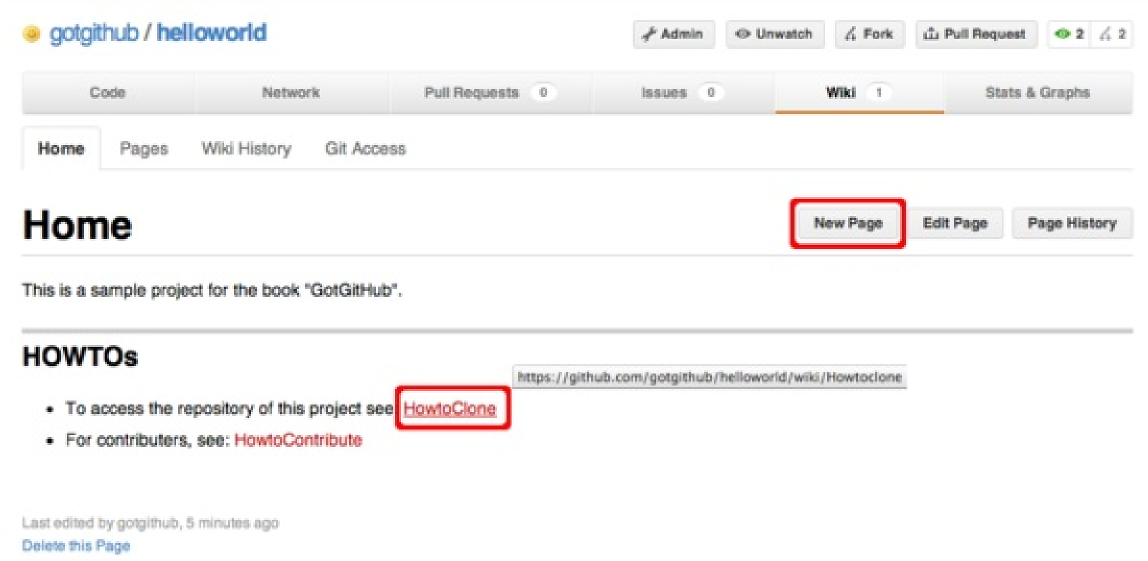


图4-81：页面中的维基链接

如图4-81所示，点击页面中指向不存在维基页面的链接，会自动开启创建新页面的对话框，如图4-82所示。

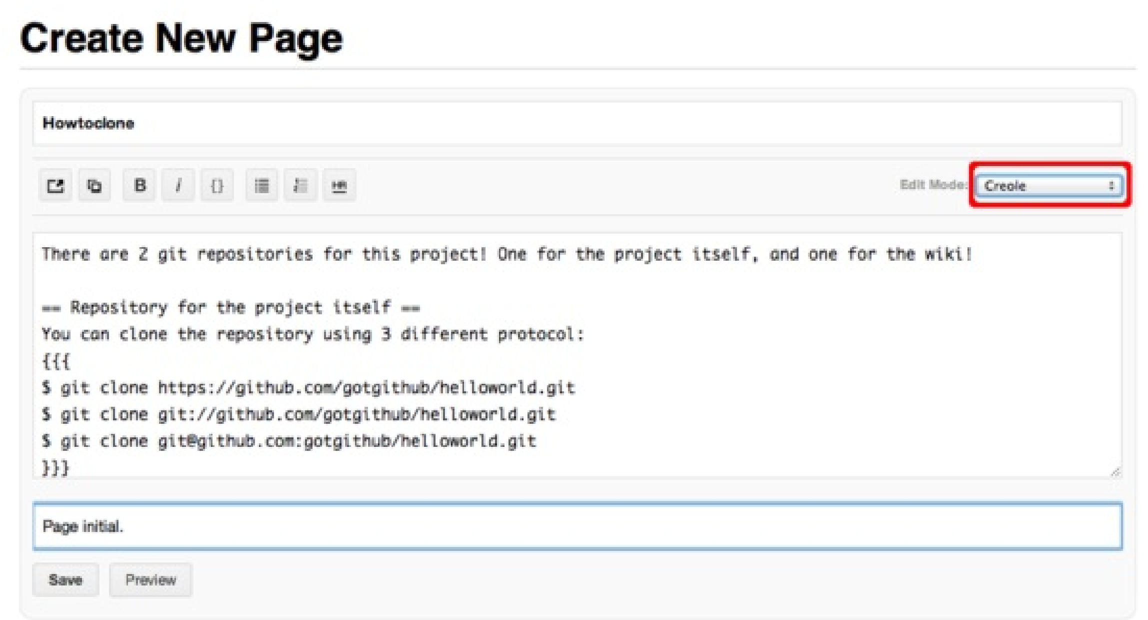


图4-82：创建新维基页面

输入维基页面的内容，然后填写提交说明，点击“Save”按钮，保存新页面。生成的新页面如图4-83所示。

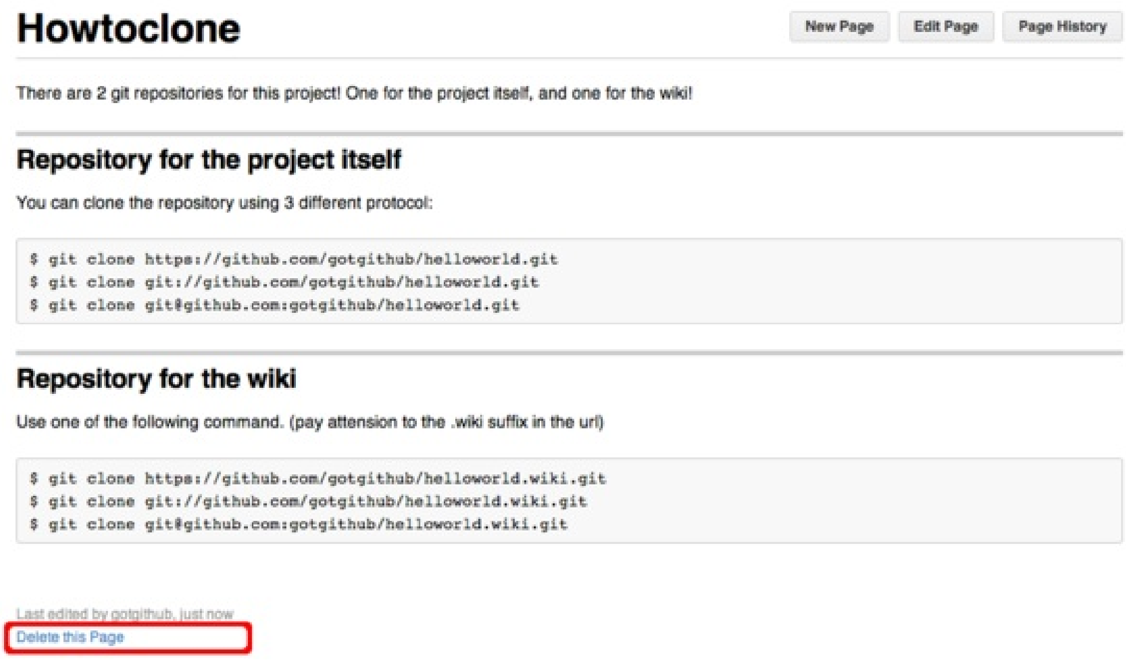


图4-83：生成的新页面

如果当前用户对页面具有写权限，则在页面左下角会看到一个删除本页面的链接。点击“Delete this page”链接并经确认后会删除页面。然后继续在维基中操作，如创建另外一个新页面HowtoContribute。

如果对之前删除页面Howtoclone的操作后悔，可以通过下面方法找回。

- 访问菜单中的“Wiki History”项，显示整个维基的修订记录（不是某个页面的修订记录）。如图4-84所示。

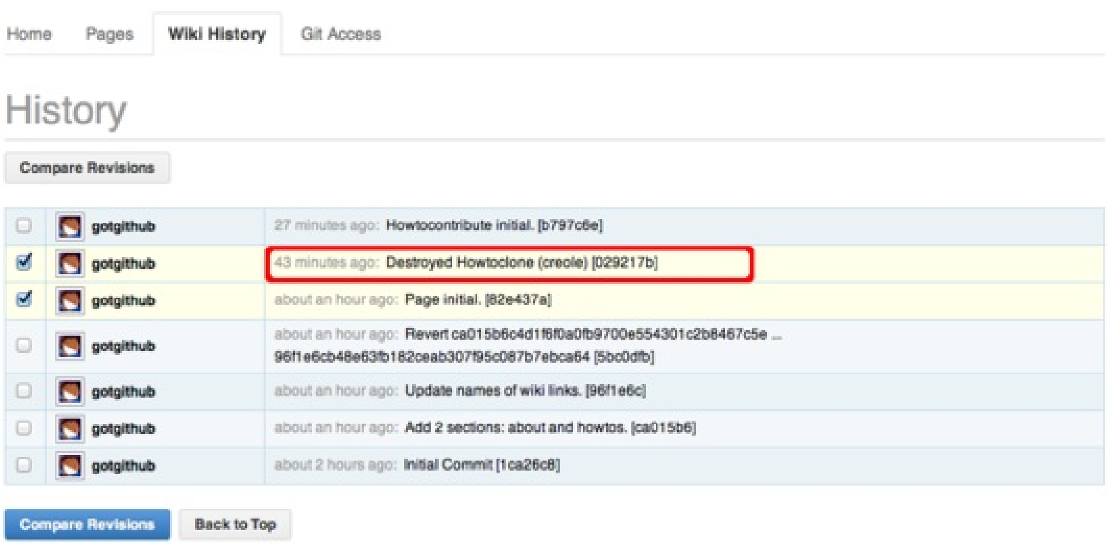


图4-84：维基修订记录

- 从图4-84可见上面第二条记录就是删除HowtoClone页面的操作，选择该记录及前一次记录，执行版本比较，如图4-85所示。

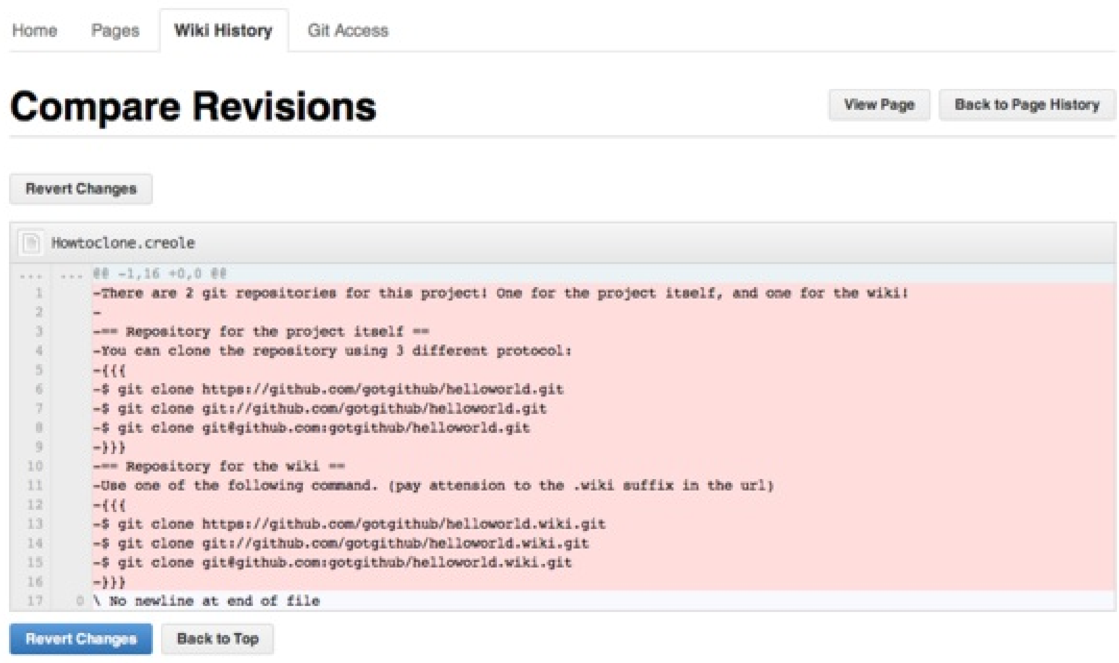


图4-85：版本比较

- 点击“Revert Changes”按钮，可以取消页面删除动作。更新后的维基修订历史如图4-86所示。

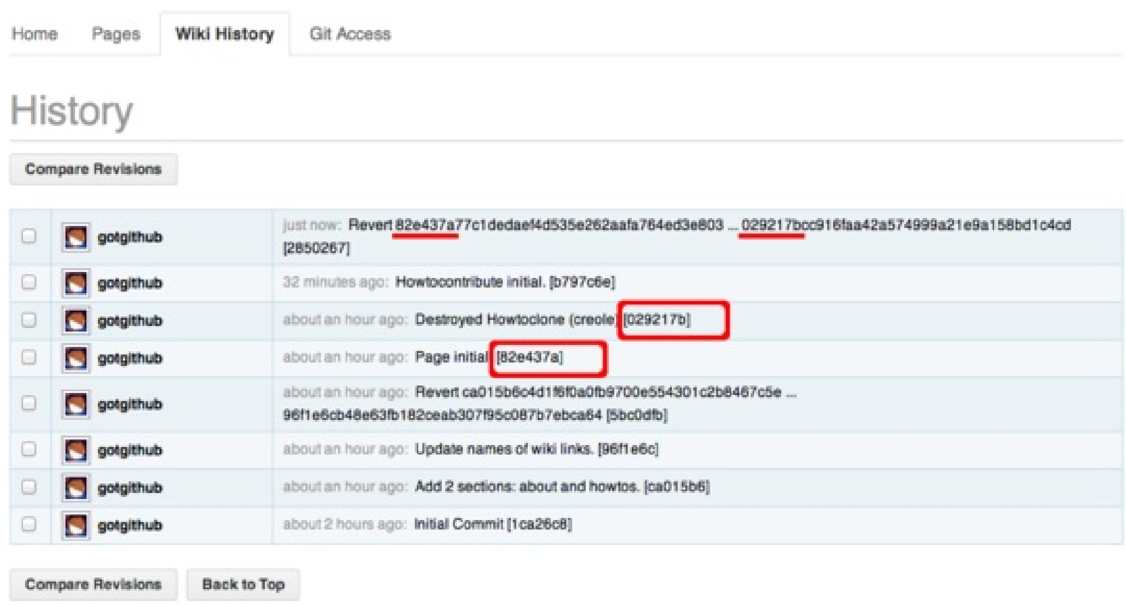


图4-86：还原修订后的维基修订记录

- 查看维基页面列表，可以看到页面Howtoclone已经被找回。

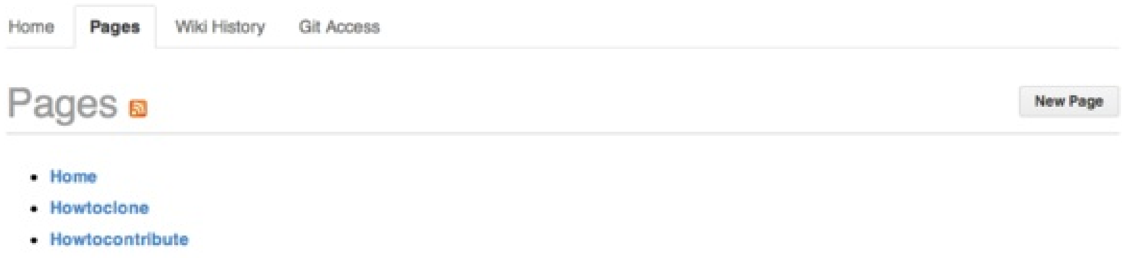


图4-87：维基页面列表

4.6.3. 维基与Git

随着对GitHub维基的深入使用，可能会遇到下面的问题：如何嵌入图片？多人编辑时如何避免冲突？解决这几个问题的办法就是用Git操作维基。在之前查看维基修订历史，以及进行版本间比较时可能已经看出和Git是如何的相似，实际上GitHub的维基页面就是用Git版本库实现的。

在维基页面访问菜单中的“Git Access”项，会看到用Git访问维基页面的方法。如图4-88所示。

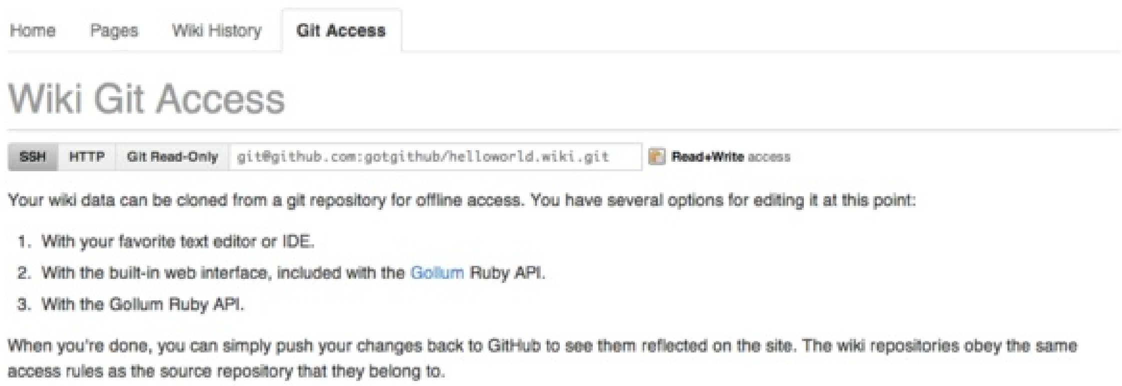


图4-88：用Git访问维基

对于项目gotgithub/helloworld来说，用Git克隆其维基，用如下命令：

```
$ git clone git@github.com:gotgithub/helloworld.wiki.git
```

进入到刚刚克隆的helloworld.wiki工作区中，查看包含的文件，会看到有三个文件。

```
$ cd helloworld.wiki
$ ls
Home.md            Howtoclone.creole  Howtocontribute.md
```

三个文件对应于三个维基页面，文件名就是维基的页面名称，而扩展名对应于采用的维基语法。以.md扩展名结尾的页面采用Markdown语法，而以.creole结尾的文件采用Creole标准维基语法。

下面就通过Git在维基版本库中添加一个图片。添加图片的操作只通过GitHub维基的Web界面是很难实现的，而使用Git则易如反掌。

- 创建一个名为images目录。这个目录并非必须，只是为了易于管理。

```
$ mkdir images
$ cd images
```

- 在images目录中添加图片。

  下面的操作从GitHub官方版本库中下载图片octocat.png并进行适当缩放。

```
$ wget https://github.com/github/media/raw/master/octocats/octocat.png
$ mogrify -resize '200' octocat.png
```

- 将图片添加到暂存区并提交。

```
$ git add octocat.png
$ git commit -m "add sample image."
```

- 将本地提交推送到GitHub远程版本库。

```
$ git push
```

完成推送后，访问下面的网址可以看到刚刚上传的图片：

https://github.com/gotgithub/helloworld/wiki/images/octocat.png

接下来在维基页面中引用图片。嵌入图片的Markdown语法是：。当然可以通过编辑本地版本库gotgithub/helloworld.wiki.git中的文件，但通过GitHub维基编辑界面嵌入图片无需记忆复杂的语法。如图4-89所示。

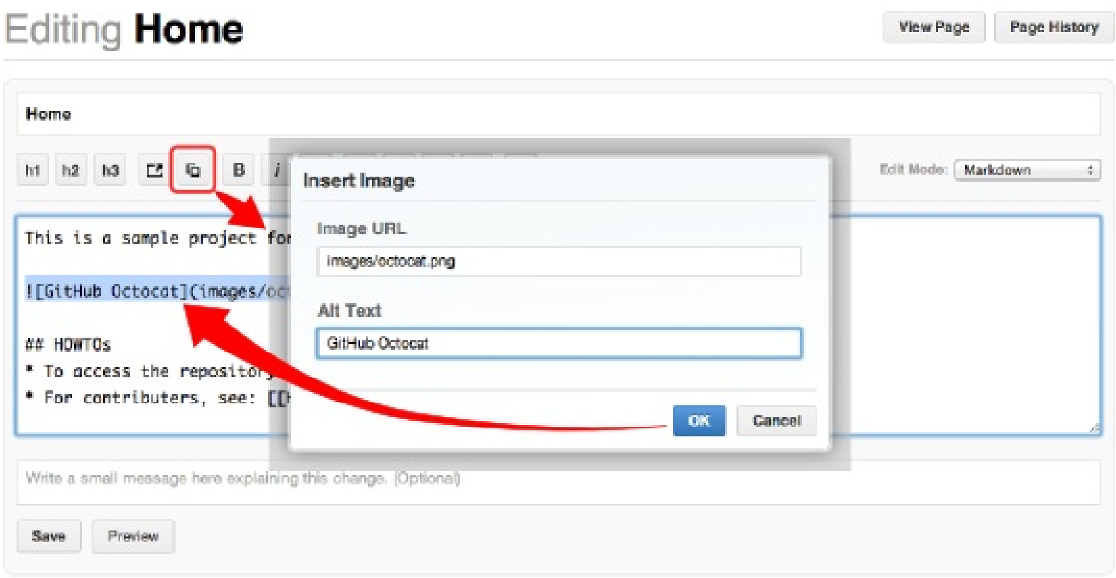


图4-89：在维基页面中嵌入图片

点击“Preview”按钮，可以在保存前查看效果。在图4-90所示的预览界面中可以看到修改后的效果。

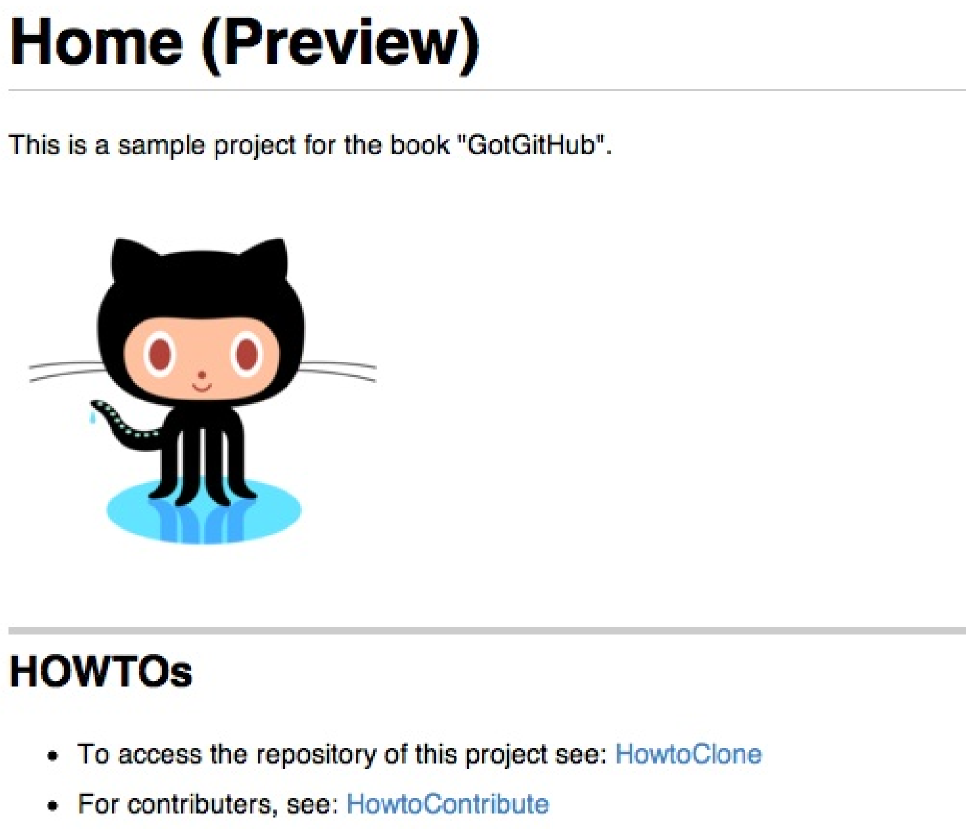


图4-90：预览效果

多人同时编辑一个维基页面会引起冲突，先提交的用户会成功，其他用户的编辑界面马上会显示冲突警告，并且保存按钮也被置灰，如图4-91所示。

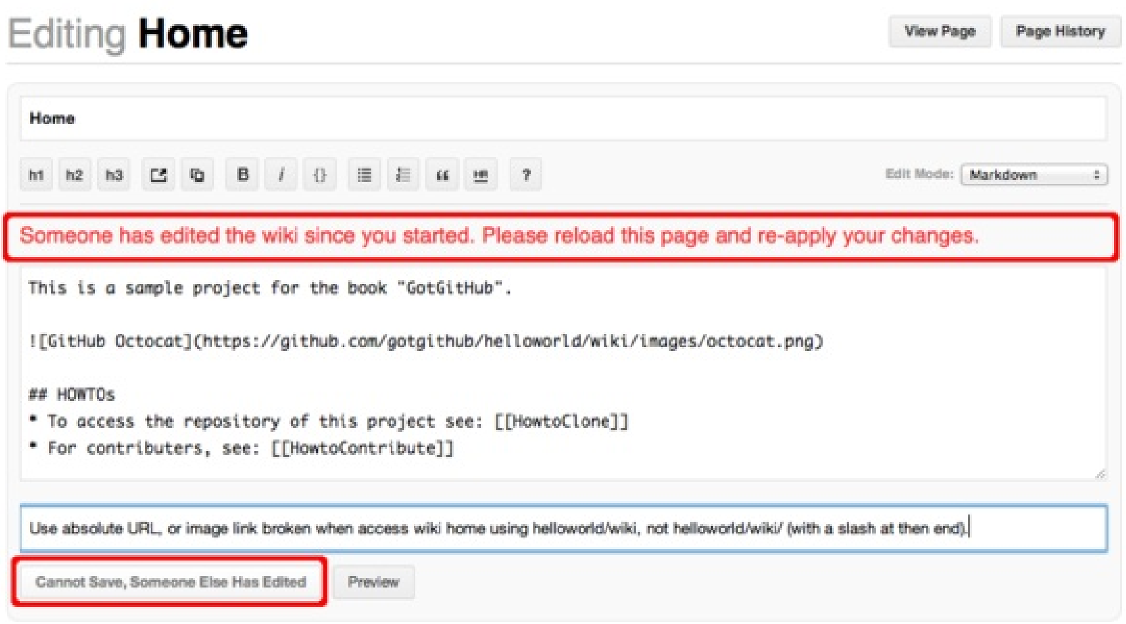


图4-91：编辑冲突

GitHub的维基编辑界面没有提供冲突解决的工具，而利用Git本身强大的冲突解决功能可以很容易地解决这一难题。

例如用户gotgithub编辑维基首页Home遇到编辑冲突，为防止数据丢失，先将编辑框中的维基文本拷贝并粘贴到一个临时文件中，如文件中/path/to/draft.md。然后进行如下操作将draft.md中内容合并到维基页面中。

- 如果本地已经克隆维基版本库，则执行下面命令更新。

```
$ cd helloworld.wiki
$ git pull
```

- 如果没有，则克隆维基版本库。

```
$ git clone git@github.com:gotgithub/helloworld.wiki.git
$ cd helloworld.wiki
```

- 用Git命令查看版本库的历史，以便找出发生冲突的原始版本。

  从下面的输出可以看出我们编辑的版本是基于提交fbb4bb4，由于用户incredible先于我们完成了对维基页面的修改以致发生了冲突。

```
$ git log -3 --pretty=short
commit 5ff5d998bb6cf99337813915282df94701d17ea0
Author: incredible

    Add a note as image link broken if url without a end slash.

commit fbb4bb4f330bacf765d51736359b0a3e81ed945b
Author: gotgithub

    Insert image in page.

commit 94182c2b57ebce1f1bf8a310f78df87ae8e8219a
Author: gotgithub

    add sample image.
```

- 基于提交fbb4bb4建立分支，如分支mywiki。

```
$ git checkout -b mywiki fbb4bb4
```

- 将保存的draft.md覆盖欲修改的文件，如Home.md。

```
$ cp /path/to/draft.md Home.md
```

- 提交修改。

```
$ git add -u
$ git commit -m "Use absolute image link."
```

- 切换到master分支。

```
$ git checkout master
```

- 合并我们在mywiki分支的修改。

```
$ git merge mywiki
Auto-merging Home.md
CONFLICT (content): Merge conflict in Home.md
Automatic merge failed; fix conflicts and then commit the result.
```

- 调用图形工具解决冲突。

```
$ git mergetool
```

- 提交并查看合并后的提交关系图。

```
$ git commit -m "merge with incredible's edit."
$ git log --oneline --graph -4
*   d33b55a merge with incredible's edit.
|\
| * 121c3b2 Use absolute image link.
* | 5ff5d99 Add a note as image link broken if url without a end slash.
|/
* fbb4bb4 Insert image in page.
```

- 查看用户incredible的修改。

```
$ git show --oneline HEAD^1
5ff5d99 Add a note as image link broken if url without a end slash.
diff --git a/Home.md b/Home.md
index 6ada8e8..0bca3ec 100644
--- a/Home.md
+++ b/Home.md
@@ -1,5 +1,7 @@
 This is a sample project for the book "GotGitHub".

+**Note**: if can not see the following image, add a slash('/') at the end of the URL.
+
 ![GitHub Octocat](images/octocat.png)

 ## HOWTOs
```

- 查看用户gotgithub的修改。

```
$ git show --oneline HEAD^2
121c3b2 Use absolute image link.
diff --git a/Home.md b/Home.md
index 6ada8e8..cdb9167 100644
--- a/Home.md
+++ b/Home.md
@@ -1,6 +1,6 @@
 This is a sample project for the book "GotGitHub".

-![GitHub Octocat](images/octocat.png)
+![GitHub Octocat](https://github.com/gotgithub/helloworld/wiki/images/octocat.png)

 ## HOWTOs
 * To access the repository of this project see: [[HowtoClone]]
```

- 将本地合并后的版本库推送到GitHub。

```
$ git push
```

再来看看推送后GitHub的维基修订历史，和本地版本库看到的历史是一致的，如图4-92所示。

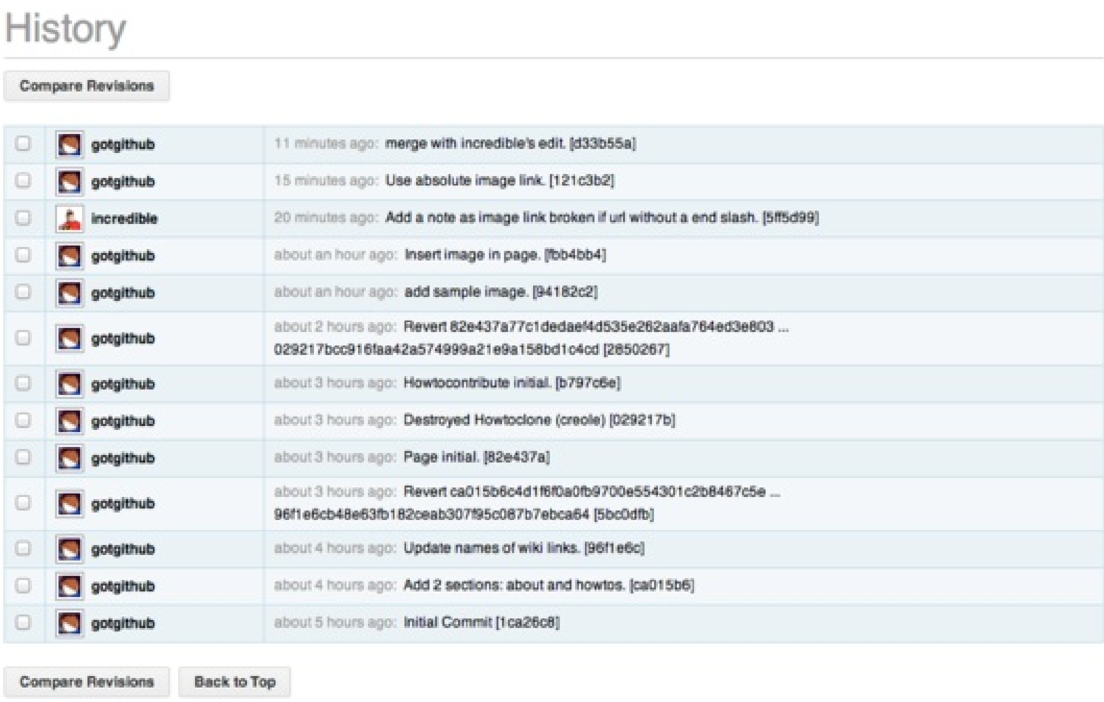


图4-92：推送后的维基修订历史

GitHub维基背后的引擎名为Gollum，GitHub已将其开源，项目网址：https://github.com/github/gollum 。安装Gollum，在克隆的维基版本库中运行gollum就可以在本地启动维基服务。

5. 付费服务

GitHub神奇的协同工具使得开源项目的创建和协同更加简单、高效。有些人可能会提出疑问：能否把GitHub用于私有项目呢？即能否只允许指定的用户访问项目和版本库，而其他人不能访问呢？能否在企业内部架设一个一模一样的GitHub服务呢？GitHub针对这些需求提供了解决方案，这些解决方案需要或多或少地付出一定费用。

5.1. GitHub收费方案

访问网址 https://github.com/plans 可以看到GitHub提供的不同的服务方案列表。

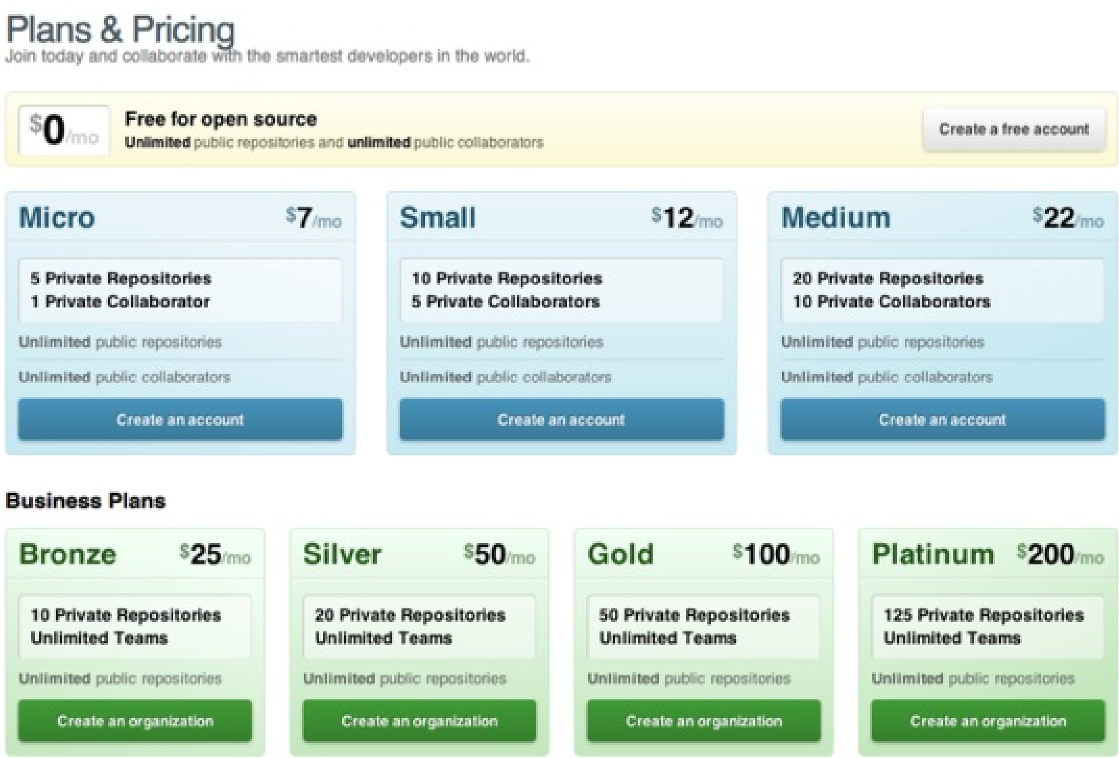


图5-1：GitHub服务方案列表

图5-1中显示了GitHub的三类（8种）服务方案：

- 第一类是免费方案。免费用户账号可以创建任意数量的开放式项目（版本库），并且可以为开放式项目设置任意数量的协同者。
- 第二类是需要付费的个人账号方案。付费的个人账号允许托管私有版本库，即可以创建只有自己及指定的私有协同者才能够访问的版本库，而其他人不能访问。根据允许创建的私有版本库数量及私有版本库协同者数量，提供了三种收费标准（7美元/月、12美元/月 和 22美元/月）。
- 第三类是需要付费的组织账号方案。使用付费的组织账号，可以突破私有项目的协同者数量限制，并使用更易管理的团队（Team）对项目进行授权。关于如何通过团队配置授权参见“4.3. 组织和团队”。组织账号的付费标准较个人账号更高（有25美元/月、50美元/月、100美元/月 和 200美元/月），但同时也可以创建更多的私有版本库和拥有更大的托管空间。

用户可以随时升级或降级自己在GitHub上的服务方案。点击菜单中的“Account Settings”可以看到当前所选方案，如图5-2所示。



图5-2：用户所选方案及状态

点击图5-2中的“Change plan”按钮，进入到更换GitHub服务方案页面，如图5-3所示。

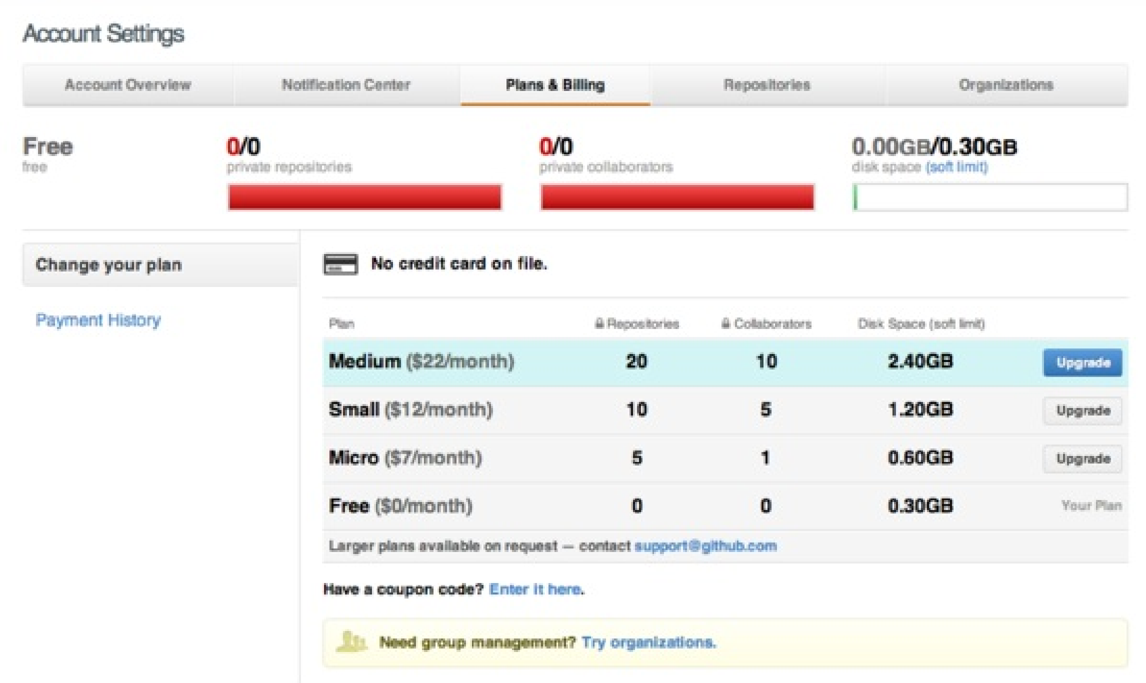


图5-3：更换方案

选择适合的收费方案并付款后，即可完成服务方案的升级。

当 gotgithub 用户升级为付费账号后，创建新版本库时就可以通过新的选项创建私有版本库了。即在创建版本库时，如果不选择默认的“Anyone”，而是选择“Only the people I specify”可以创建私有版本库，如图5-4所示。

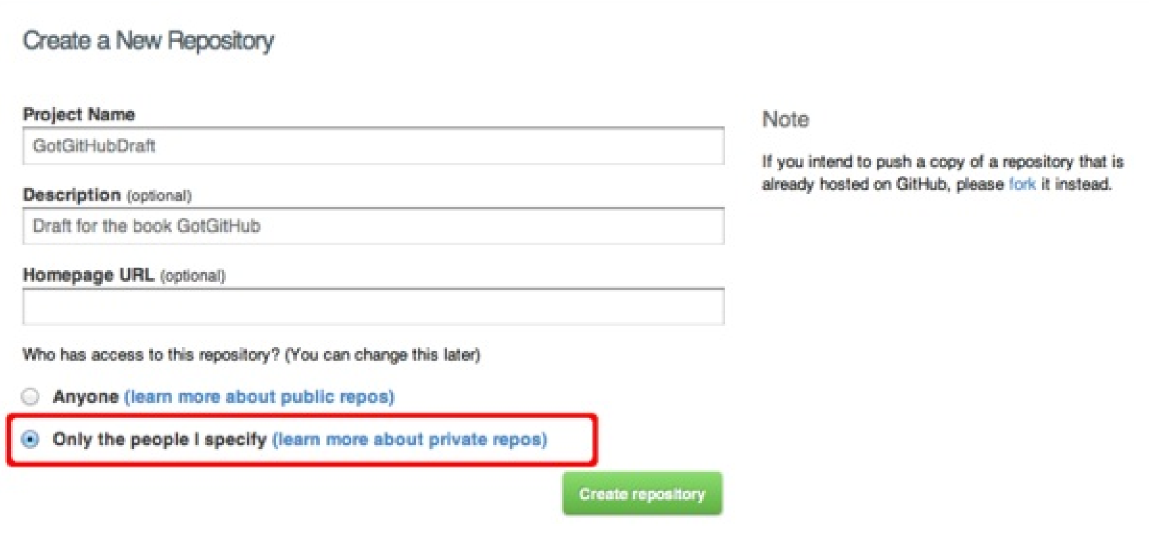


图5-4：创建私有版本库

通过版本库的管理界面，可以随时将版本库的状态在公开和私有之间切换，如图5-5所示。

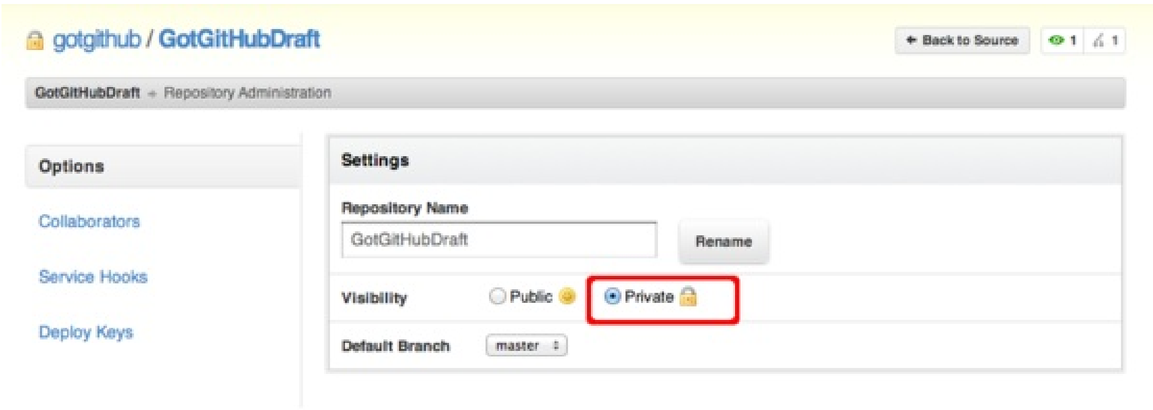


图5-5：私有版本库管理界面

付费账号的公开版本库没有协同者数量上的限制，但是私有版本库却存在协同者数量上的限制。如图5-6所示，当私有版本库的协同者数量超出所选GitHub付费方案的限额后，会显示“OVERLIMIT”的警告，不过超出限额的协同者依然可以操作私有版本库。

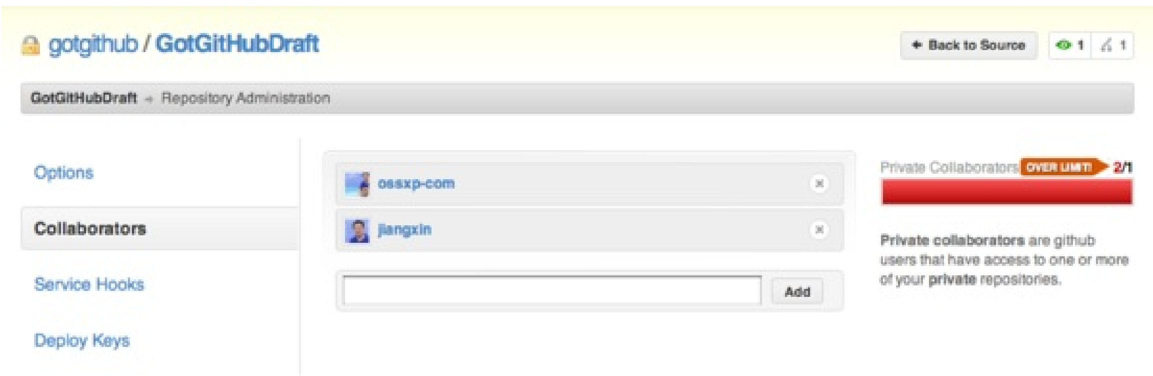


图5-6：添加私有协同者

组织是一类特殊的不能登录的用户账号。如果要对组织账号进行配置，需要先以组织所有者的用户账号登录，再通过切换上下文的方式访问组织账号。图5-7就是以gotgithub用户账号登录后，切换到 GotGitOrg 组织账号的管理界面。



图5-7：团队账号的所选方案及状态

在图5-7所示的组织账号管理界面中显示了组织账号当前的GitHub方案，点击其中的“Change plan”按钮，显示如图5-8所示界面，可对组织账号的GitHub方案进行升级或降级。

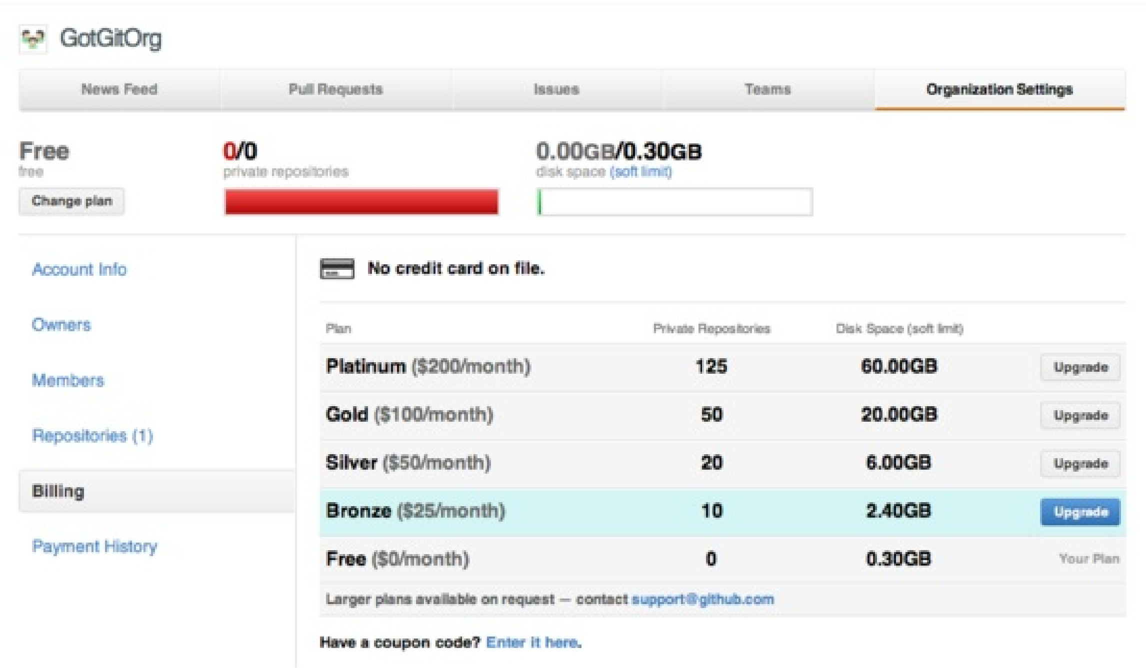


图5-8：团队账号更换方案

为组织账号选择一个付费方案后，就可以在组织的账号下创建私有版本库，并以团队方式管理该私有版本库的授权。图5-9就是一个私有版本库GotGitOrg/NonPublicRepo的设置界面。

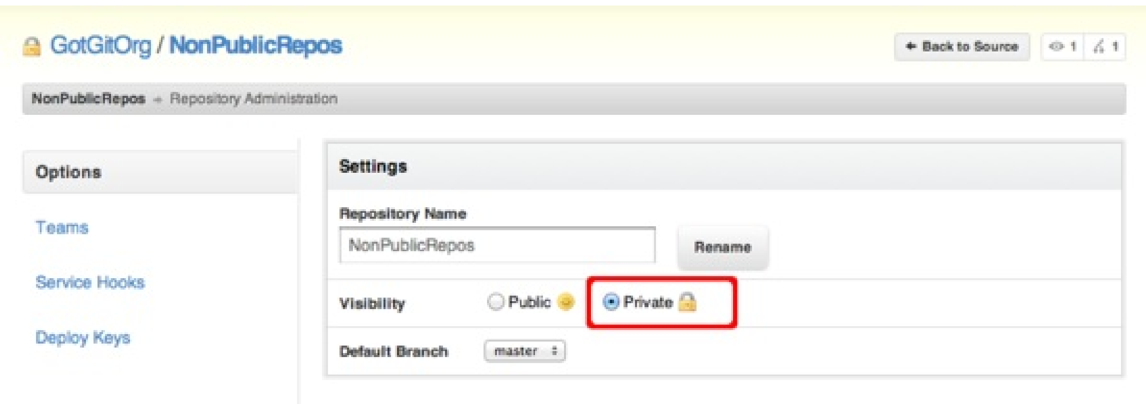


图5-9：团队的私有版本库设置

5.2. GitHub企业版

出于隐私或法律原因而不能将代码托管到第三方平台的企业，可能希望在企业内部架设专有的GitHub服务，能做到么？答案就是GitHub企业版（GitHub Enterprise）。

网址：https://enterprise.github.com/。

GitHub企业版搭建在企业本地网络中，因此企业拥有对版本库和项目完整的控制权限。GitHub企业版包含了GitHub上所有的好东西：提交历史、代码浏览、比较视图、Pull Requests、问题追踪、维基、Gists、组织和团队管理、强大的API，和漂亮的界面，因此会使用GitHub就会使用GitHub企业版。此外企业版还增加了对LDAP和CAS支持功能，以便和企业现有的认证系统整合等等[1]。

GitHub企业版不像它的前身GitHub:FI(GitHub:FI是GitHub Firewall Install的缩写，含义为在企业的防火墙内部架设GitHub服务，现已升级为GitHub Enterprise。)那样通过下载软件包进行安装和部署，而是提供基于虚拟机的解决方案。即GitHub企业版以OVF虚拟机文件格式发布，可以运行在多种虚拟机平台，如：VMWare、VirtualBox、Oracle VM、Red Hat Enterprise Virtualization和IBM POWER。使用OVF格式让GitHub企业版的部署更加轻松。

GitHub企业版根据用户数量收取年费，入门级的价格为20用户每年5,000美金。如果用户数少，建议采用付费的GitHub托管账号。购买更多用户许可，访问GitHub企业版网站，那儿有一个报价生成器，如图5-10所示。

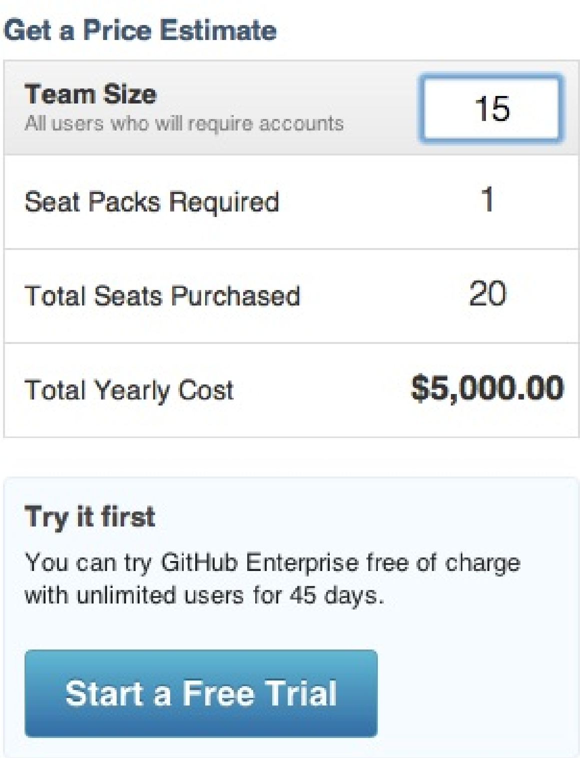


图5-10：GitHub企业版报价生成器

6. GitHub副产品

GitHub最核心的产品是Git版本库（即项目）托管，此外GitHub还提供一些副产品（Side Project），通过附加的服务或技术提供了更多有趣的功能。例如提供数据粘贴的Gist网站，对其他版本控制工具如SVN和Hg的支持，各具特色的客户端工具，求职网站，销售纪念品的GitHub商店等等。

6.1. GitHub:Gist

在GitHub网站的导航条上就有Gist子网站的链接： https://gist.github.com/ ，在本节我们就揭开其面纱。



图6-1：GitHub上的Gist链接

Gist作为一个粘贴数据的工具，就像 Pastie 网站[1]一样，可以很容易地将数据粘贴在Gist网站中，并在其他网页中引用Gist中粘贴的数据。作为GitHub的一个子网站，很自然地，Gist使用Git版本库对粘贴数据进行维护，这非常酷。

6.1.1. 数据的粘贴和引用

进入Gist网站的首页，就会看到一个大大的数据粘贴对话框，如图6-2所示。

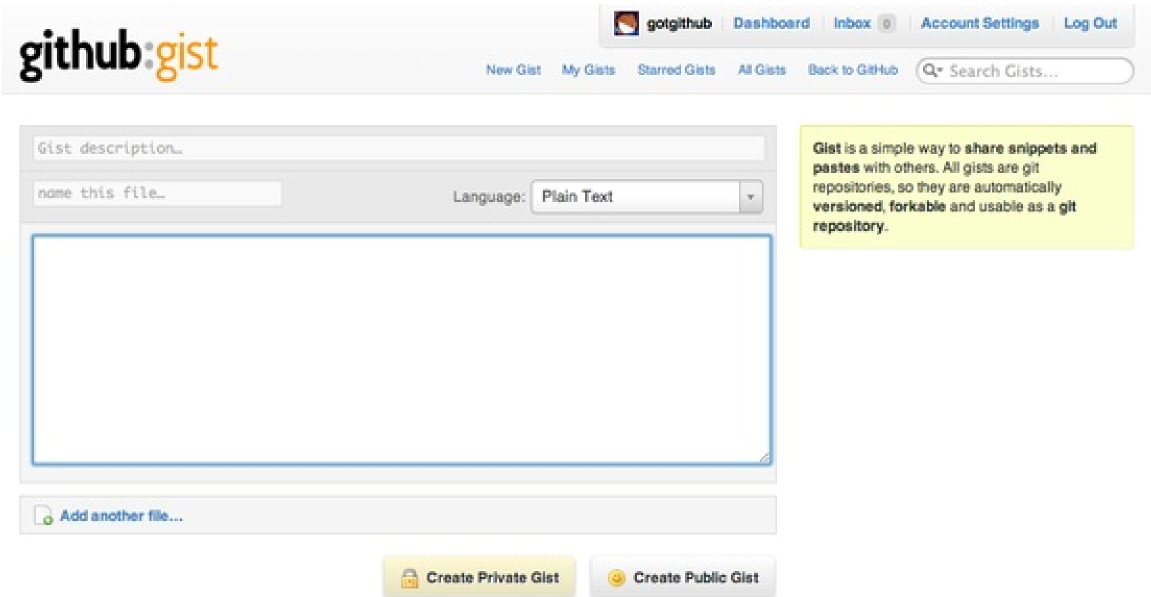


图6-2：Gist网站首页

只要提供一行简单的描述、文件名，并粘贴文件内容，即可创建一个新的粘贴。创建新的粘贴时，如果不指定文件名，将由系统自动指派。如图6-3所示。

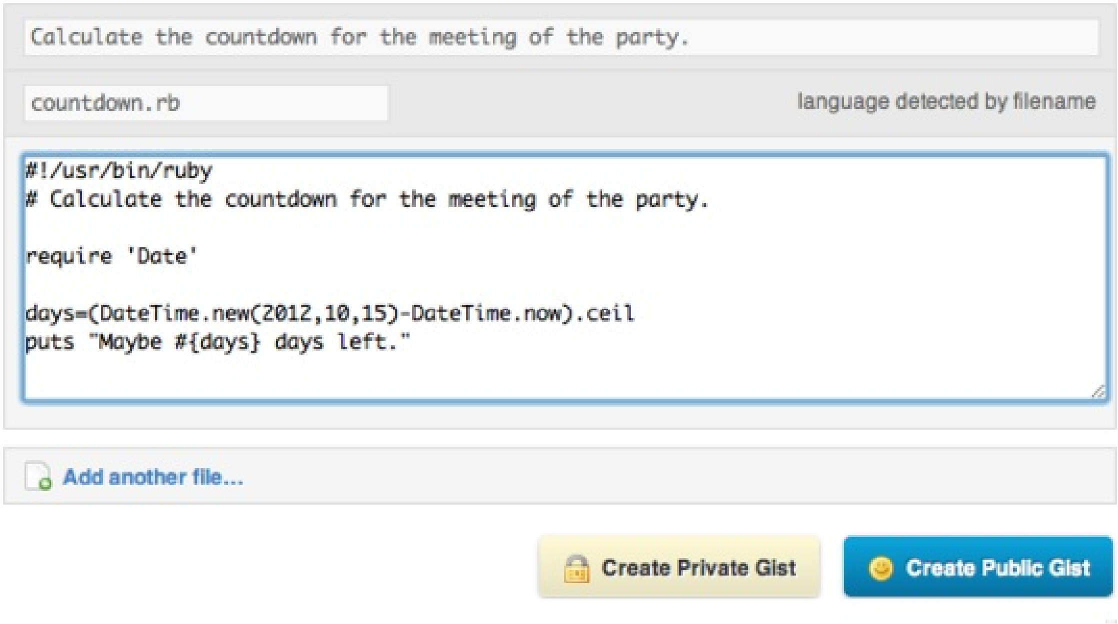


图6-3：创建新的Gist

每一个新的粘贴称为一个Gist，并拥有唯一的URL。如果选择创建公开的Gist，URL中将使用顺序递增的ID号，如本例创建的Gist的URL地址为：https://gist.github.com/1202870。

若选择创建私有Gist，URL中则采用20位十六进制数字的ID，例如私密Gist：https://gist.github.com/78d67164131ec9e08dfe。需要指出的是，私有Gist的私密性并不像GitHub私有版本库的私密性那么强，只是其URL没有在用户Gist列表中列出，也不能通过Gist网站搜索到而已。如果用户知道某私密Gist的URL地址，同样可以访问、克隆该私密Gist，甚至创建基于该Gist创建分支（fork）。

当一个粘贴创建完毕后，会显示新建立的Gist页面，如图6-4所示。

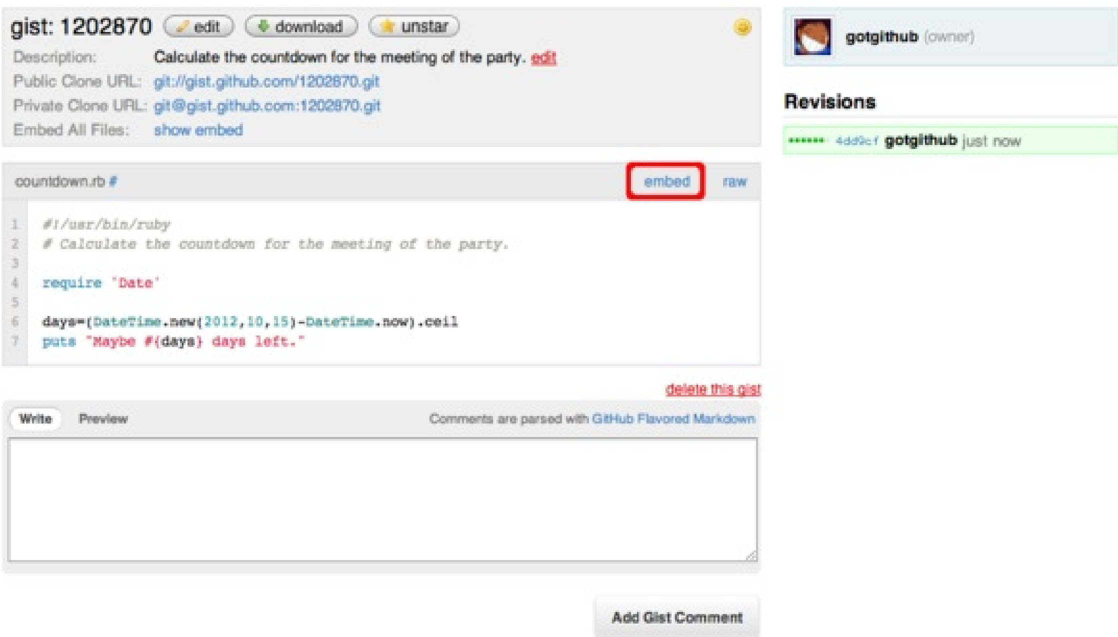


图6-4：新创建的Gist

点击其中的“embed”（嵌入）按钮，就会显示一段用于嵌入其他网页的JavaScript代码，如图6-5所示。

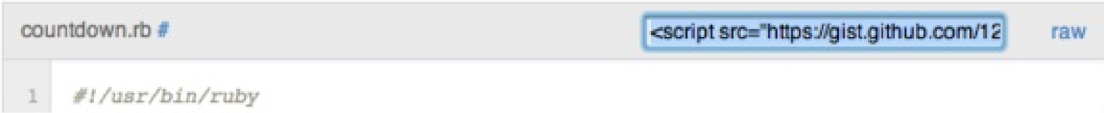


图6-5：显示嵌入Java

对应的嵌入JavaScript代码如下：

```
<script src="https://gist.github.com/1202870.js?file=countdown.rb"></script>
```

将上面的JavaScript代码嵌入到网页（如博客[2]）中，即可在相应的网页中嵌入来自Gist的数据，并保持语法加亮等功能，如图6-6所示。



图6-6：博客中引用Gist数据

6.1.2. Gist背后的Git库

创建的每一个Gist的背后都对应着一个Git版本库。例如之前创建的ID为1202870的Gist对应的Git版本库，可以使用两种协议进行访问：

- Git协议：git://gist.github.com/1202870.git
- SSH协议：git@gist.github.com:1202870.git

可以通过Git命令克隆和操作该版本库。

- 克隆该Gist对应的版本库。

```
$ git clone git@gist.github.com:1202870.git
$ cd 1202870
```

- 查看修改日志。每一次对Gist中文件的修改对应于一次提交。

```
$ git log
commit 993d28a1319eca314ab2e3f4c46882cf328e5ff9
Author: GotGitHub <gotgithub@gmail.com>
Date:   Thu Sep 15 15:41:10 2011 +0800

commit 4dd9cfd54e1522d0b62d92dd5f705a61e3fe8778
Author: GotGitHub <gotgithub@gmail.com>
Date:   Thu Sep 8 00:46:50 2011 -0700
```

- 查看最近一次更改。

```
$ git show HEAD
commit 993d28a1319eca314ab2e3f4c46882cf328e5ff9
Author: GotGitHub <gotgithub@gmail.com>
Date:   Thu Sep 15 15:41:10 2011 +0800

diff --git a/countdown.rb b/countdown.rb
index a9d747b..9045738 100644
--- a/countdown.rb
+++ b/countdown.rb
@@ -4,4 +4,8 @@
 require 'Date'

 days=(DateTime.new(2012,10,15)-DateTime.now).ceil
-puts "Maybe #{days} days left."
\ No newline at end of file
+if days >= 0
+  puts "Maybe #{days} days left."
+else
+  puts "Passed for #{days.abs} days."
+end
\ No newline at end of file
```

Gist网站并没有像GitHub网站那样对于Git版本库提供完整的、近乎复杂的操作界面和工作流支持，而只提供了最基本的操作界面。如图6-7所示。



图6-7：Gist版本库简易操作界面

在这个简易的Git版本库操作界面中，左侧是版本库的简介、文件预览以及在线编辑、下载、加注星标(对感兴趣的Gist进行收藏，参见博客 https://github.com/blog/673-starring-gists)、版本库分支(访问他人创建的Gist时，提供分支功能按钮。)等相关操作按钮。若以Gist创建者登

录，会在右侧看到他人基于该Gist创建分支的情况，但是并不提供GitHub才有的Pull Request等功能。在界面的右侧还显示了Gist修订历史，和之前通过git log命令从Git版本库看到的一样。

6.1.3. Greasemonkey

Gist除了被用于粘贴数据（如代码块）并在网页中引用之外，还被用户挖掘出了新的应用模式，例如用作Greasemonkey脚本的维护[5]。

Greasemonkey[6]或类似插件为浏览器提供用户端JavaScript扩展功能，最早出现于FireFox浏览器中。其他浏览器也陆续增加了对用户端JavaScript的支持，如Safari的 NinjaKit[7]插件，IE的Trixie[8]插件，以及Chrome的Greasemetal插件(版本4之后的Chrome内置了Greasemonkey类似功能，无需额外插件。)。关于如何在浏览器中安装并启用相应的插件，参照相关插件网站的介绍，在此不做过多叙述。

当浏览器安装了 Greasemonkey 或类似插件之后，当访问扩展名为.user.js的URL时，会将该URL指向的JavaScript脚本安装在浏览器中，当访问指定的网址时会自动调用相应的JavaScript脚本，修改相关网页内容或添加特效等等。

我针对《Git权威指南》官网的测试网页写了一个Greasemonkey示例脚本，可以展示用户端JavaScript的魔法，这个用户端JavaScript脚本保存在Gist中：https://gist.github.com/1084591，如图6-8所示。

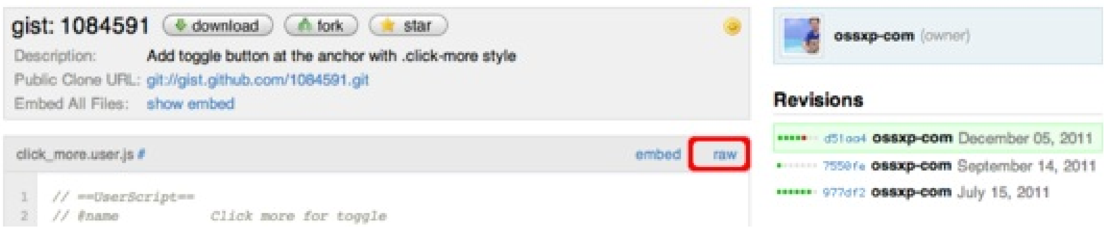


图6-8：保存Greasemonkey用户端脚本的Gist

该Greasemonkey脚本的文件名为click_more.user.js，该文件的文件头使用特殊的注释语句为Greasemonkey提供相关的安装和注册信息，内容如下（为方便描述添加了行号）：

```
1  // ==UserScript==
2  // @name            Click more for toggle
3  // @namespace       gotgit
4  // @description     Add a toogle effect at the location where anchor with a click-more
5  // @include         http://www.worldhello.net/gotgit/demo*
6  // @include         http://gotgit.github.com/gotgit/demo*
7  // @include         http://www.ossxp.com/doc/gotgit/demo*
8  // @require         http://code.jquery.com/jquery-1.6.2.min.js
9  // ==/UserScript==
```

其中第5、6、7行三条include语句限定了此用户端JavaScript脚本的应用范围，即只针对指定的URL（使用通配符）执行该脚本。第8行设定脚本依赖，即该脚本依赖jQuery，会在运行前到指定的URL地址加载jQuery脚本。

在安装该脚本前，先用浏览器访问网址http://www.worldhello.net/gotgit/demo.html，看看不加载用户端JavaScript脚本时网页的模样。该网页中包含一个长长的网上书店列表，如图6-9所示。



图6-9：应用用户端JavaScript脚本前的网页内容

接下来开始安装该用户端JavaScript脚本。安装非常简单，只要点击图6-8的Gist当中的脚本文件对应的“raw”链接，即点击脚本文件原始内容链接[10]即可开启安装。这是因为该URL以.user.js结尾，会被Greasemonkey（或类似插件）识别并安装，如图6-10是Greasemonkey弹出的用户端脚本安装界面。



图6-10：安装用户端JavaScript脚本

用户端脚本安装完毕后，再访问同样的测试网页http://www.worldhello.net/gotgit/demo.html，会发现网页中出现了一个名为“更多”的可点击链接，长长的网上书店列表不见了。如图6-11所示。



图6-11：应用用户端JavaScript脚本后的网页内容

如果查看网页源码，会发现该网页中根本没有包含和调用任何JavaScript脚本，只是在页面源码中包含着一个没有任何实质输出的标签：

```
<p><a class="click-more"></a></p>
```

实际上正是这个特殊的标签被Greasemonkey所加载的用户端脚本识别，为HTML网页添加了特效。

6.1.4. 命令行操作Gist

GitHub开发者还写了一个名为gist的命令行工具对Gist进行操作，地址见 https://github.com/defunkt/gist 。

该工具使用Ruby开发，对两个特定的Git风格的配置变量进行如下设置后，即可实现在命令行中自动以特定用户身份登录操作Gist。

```
$ git config --global github.user "your-github-username"
$ git config --global github.token "your-github-token"
```

其中github.token中保存的是用户的API TOKEN，这在“2.1 创建GitHub账号”一节有过介绍。

使用gist命令行工具创建新的Gist非常简单。

- 创建包含一个文件（如script.py）的Gist，使用如下命令。

```
$ gist script.py
```

- 创建包含多个文件的Gist，使用类似如下的命令。

```
$ gist script.js notes.txt
```

如果对命令行操作方式感兴趣，参考gist工具网站的README文件。

6.2. 其他版本控制工具支持

如果曾经有人对 GitHub 只提供唯一的版本库格式（Git）托管表示过怀疑的话，那么今天看到 GitHub 对其他版本控制工具提供的愈发完善的支持，争议应该烟消云散了吧。

6.2.1. 用SVN操作GitHub

2008年4月1日，GitHub宣布推出基于SVN的SVNHub网站，后证实这是一个愚人节玩笑[1]。2010年愚人节，类似消息再起，可这一次不再是玩笑[2]。即对于GitHub上的每一个Git版本库，现在都可以用SVN命令进行操作。更酷的是 SVN 版本库使用的是和 Git 版本库同样的地址[3]。

例如用下面的 Git 命令访问本书的 Git 版本库，显示版本库包含的引用。其中分支master用于维护书稿，分支gh-pages保存书稿编译后的 HTML 网页用于在 GitHub 上显示。

```
$ git ls-remote --heads https://github.com/gotgit/gotgithub
ce5d3dda9b9ce8ec90def1da10181a094bea152f        refs/heads/gh-pages
c4d370b1b0bafb103de14e104ca18b8c31d80add        refs/heads/master
```

如果使用 SVN 命令访问相同的版本库地址，Git 服务器变身为一个 SVN 服务器，将Git的引用对应为 SVN 风格的分支。如下：

```
$ svn ls https://github.com/gotgit/gotgithub
branches/
trunk/
$ svn ls https://github.com/gotgit/gotgithub/branches
gh-pages/
```

SVN 支持部分检出，下面命令将整个主线trunk（相当于 Git 版本库的master分支）检出。

```
$ svn checkout https://github.com/gotgit/gotgithub/trunk gotgithub
A    gotgithub/Makefile
A    gotgithub/README.rst
...
Checked out revision 30.
```

还可以使用 SVN 命令创建分支，即相当于在 Git 版本库中创建新的引用。测试发现GitHub 尚不支持 SVN 远程拷贝创建分支，需要通过本地拷贝再提交的方式创建新分支。操作如下：

1. 为避免检出版本库所有分支过于耗时，在检出时使用--depth=empty参数。

```
$ svn checkout --depth=empty \
  https://github.com/gotgit/gotgithub gotgithub-branches
Checked out revision 30.
```

1. 进入到检出目录中，更新出trunk和branches两个顶级目录。

```
$ cd gotgithub-branches
$ svn up --depth=empty trunk branches
A    trunk
Updated to revision 30.
A    branches
Updated to revision 30.
```

1. 通过拷贝从主线trunk创建分支branches/svn-github。

```
$ svn cp trunk branches/svn-github
A         branches/svn-github
$ svn st
A  +    branches/svn-github
```

1. 提交完成分支创建。

```
$ svn ci -m "create branch svn-github from trunk"
Authentication realm: <https://github.com:443> GitHub
Username: gotgithub
Password for 'gotgithub':
Adding         branches/svn-github

Committed revision 31.
```

1. 用 Git 命令可以看到服务器上创建了一个新的同名引用，并且指向和master一致。

```
$ git ls-remote --heads https://github.com/gotgit/gotgithub
ce5d3dda9b9ce8ec90def1da10181a094bea152f        refs/heads/gh-pages
c4d370b1b0bafb103de14e104ca18b8c31d80add        refs/heads/master
c4d370b1b0bafb103de14e104ca18b8c31d80add        refs/heads/svn-github
```

下面尝试一下用 SVN 命令在新创建的分支svn-github中提交。

1. 进入到之前检出完整主线trunk的gotgithub目录，并将工作区切换为分支branches/svn-github。

```
$ cd ../gotgithub
$ svn switch https://github.com/gotgit/gotgithub/branches/svn-github
At revision 31.
```

1. 修改文件，查看工作区状态。

```
$ svn st
M       06-side-projects/040-svn.rst
```

1. 用 SVN 提交。

```
$ svn ci -m "GitHub svn client support improved. Refs: http://git.io/svn"
Sending        06-side-projects/040-svn.rst
Transmitting file data .
Committed revision 32.
```

1. 同样查看 Git 版本库的更新，会发现svn-github分支的指向已和master不同。

```
$ git ls-remote --heads https://github.com/gotgit/gotgithub
ce5d3dda9b9ce8ec90def1da10181a094bea152f        refs/heads/gh-pages
c4d370b1b0bafb103de14e104ca18b8c31d80add        refs/heads/master
64b80cb5331e28fdfb896e2ab3085779bf6ca019        refs/heads/svn-github
```

6.2.2. 用Hg操作GitHub

Hg（又称Mercurial）和 Git 一样也是一个被广泛使用的分布式版本库控制工具。如果一个熟悉 Hg 的开发者参与托管在 GitHub 上的项目，大可不必为更换版本控制工具而苦恼，GitHub 上的一个名为 hg-git[1]的开源项目可以帮上忙。

得益于 Hg 的强大的插件扩展机制，安装 hg-git 并将其注册为Hg 插件后可提供Hg操作 Git 版本库的能力。安装 hg-git 可以直接使用 **easy_install** 命令：

```
$ sudo easy_install hg-git
```

还可以直接从GitHub上下载hg-git最新代码进行安装：

```
$ curl -L -k -o hg-git.zip https://github.com/schacon/hg-git/zipball/master
$ unzip hg-git.zip
$ cd schacon-hg-git-*
$ sudo easy_install .
```

插件 hg-git 依赖于 Dulwich 项目，如果在安装过程遇到 Dulwich 无法编译，可能是因为缺乏 C 编译器，或者尚未安装 python-dev 软件包。Dulwich 是一个Python语言的 Git 实现，因此 hg-git 在运行过程中无需 Git 命令行。

和其他 Hg 插件类似，安装完毕后需要修改Hg配置文件（如文件 ~/.hgrc ）如下，以启用 hg-git 插件以及另外一个必须的 Hg 内置插件 —— bookmarks 插件。

```
[extensions]
bookmarks =
hggit =
```

对 hg-git 安装配置完毕，就可以使用 Hg 操作 Git 版本库了。

- 克隆 Git 版本库。

```
$ hg clone git://github.com/ossxp-com/hello-world.git
$ cd hello-world
```

- Git 版本库的分支转换为 Hg 版本库中的 bookmarks。

  新克隆的 Hg 版本库默认会更新到最新提交（即 tip 版本），未必处于所需的分支上。用命令**hg bookmarks**显示分支列表，命令**hg parents**显示工作区对应的版本。

```
$ hg bookmarks
   helper/master             10:2767ad9d7008
   helper/v1.x               8:994c2f0adc0b
   master                    1:dcd365e3175c

$ hg parents
修改集:      12:928384ca1e87
标签:        jx/v1.0-i18n
标签:        tip
用户:        Jiang Xin <jiangxin@ossxp.com>
日期:        Fri Dec 31 12:12:42 2010 +0800
摘要:        Translate for Chinese.
```

- 切换到所需的工作分支（如master分支）。

  用hg update -r命令切换分支。之后执行**hg bookmarks**命令会看到当前工作分支用星号标识出来。

```
$ hg update -r master
   $ hg book
      helper/master             10:2767ad9d7008
      helper/v1.x               8:994c2f0adc0b
    * master                    1:dcd365e3175c
```

- Git 的里程碑也被记录，并可被 **hg tags** 命令显示。

```
$ hg tags
   tip                                12:928384ca1e87
   jx/v1.0-i18n                       12:928384ca1e87
   jx/v2.3                            10:2767ad9d7008
   ...
```

- 使用**hg pull**命令和**hg push**命令可以实现和Git版本库的同步。
- 有的命令如**hg outgoing**可在 1.7 版本的Hg中运行正常，但对于高版本库的 Hg 存在兼容性问题。

实际上 hg-git 插件并非只针对 GitHub 的版本库，而是可以支持任意 Git 版本库包括本地 Git 版本库。为了提供对 Git 版本库的透明支持，对 Git 版本库的 URL的写法有特殊要求，即要能够从协议名称区分开 Git 版本库和默认的 Hg 版本库。

- Git协议：

  git://example.com[:port]/path/to/repo.git

- SSH协议：

  git+ssh://[user@]example.com[:port]/path/to/repo.git

- HTTP协议：

  git+http://[user@]example.com[:port]/path/to/repo.git

- HTTPS协议：

  git+

- 本地协议：

  /path/to/repo.git

6.3. 客户端工具

GitHub提供的Web服务，在客户端通常只需要浏览器及Git命令行工具就可以满足需要了。而GitHub还开发了一些客户端工具，以便用户有更好的客户端体验。

6.3.1. github:mac

GitHub专为Mac用户开发了一款图形化客户端应用github:mac，在Mac下操作GitHub更简单。软件下载地址： http://mac.github.com/ 。

github:mac 可以实现版本库克隆、查看历史、提交、分支管理、与GitHub同步等功能。图6-12展示的是提交界面，在提交界面中同时显示了变更的差异比较，用户可以挑选文件中的部分变更进行提交，显然这个操作要比在命令行中执行**git add –patch**或**git commit –patch**要更加直观。

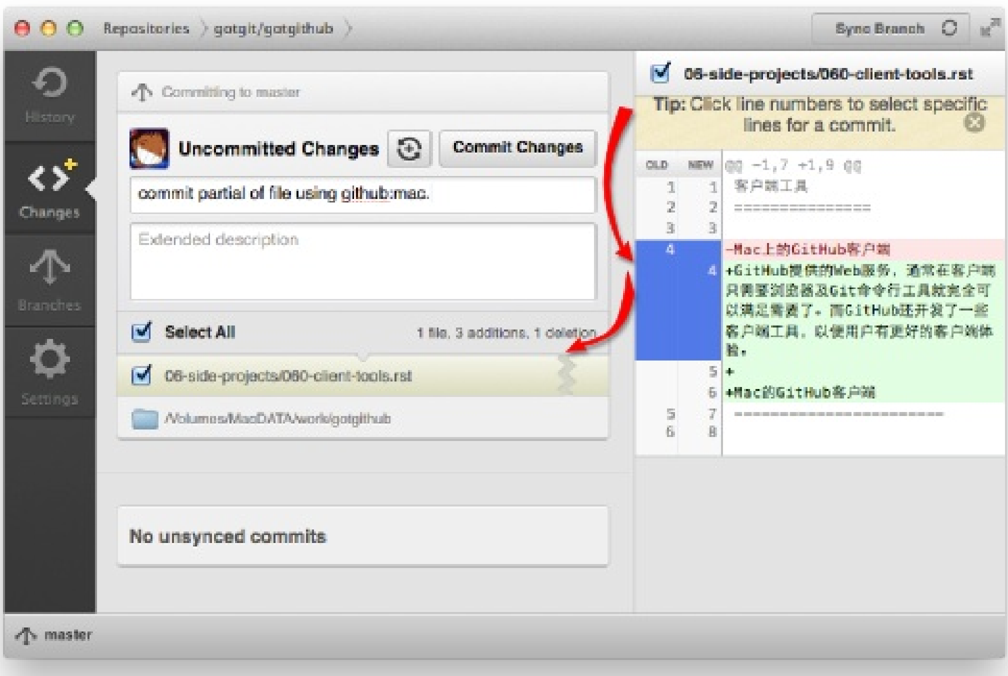


图6-12：筛选文件中的部分更改进行提交

github:mac和GitHub深度集成，当配置好关联的GitHub账号后，会自动在本地创建专用的SSH公钥/私钥对文件 ~/.ssh/github_rsa （如果该文件不存在的话），然后将公钥文件传递到GitHub网站并自动完成配置。新增的SSH公钥文件显示在GitHub网站的账号设置中，如图6-13所示：



图6-13：在GitHub上自动添加的SSH公钥

同时github:mac还在本地将新生成的私钥文件添加到ssh-agent认证代理中，这样一旦通过SSH 协议连接GitHub，首先采用该公钥/私钥对进行身份认证。用下面的命令可以查看添加到ssh-agent中的私钥文件。

```
$ ssh-add -l
2048 aa:01:4f:d2:14:ba:5f:9f:8c:dc:b5:9d:44:cd:8e:18 /Users/jiangxin/.ssh/github_rsa (RSA
```

这种透明的公钥认证管理非常酷，对于大多数只使用唯一一个GitHub账号的用户来说是非常方便的。但如果用户拥有多个GitHub账号并需要不时切换账号，这种实现却很糟糕，会导致认证错误。因为当ssh-agent认证代理缓存了私钥后，连接由文件~/.ssh/config 设置的 SSH 别名主机无法使用指定的公钥/私钥对进行认证，导致认证失败。

遇到 GitHub 账户 SSH 认证问题，可以运行下面命令清空ssh-agent缓存的私钥。

```
$ ssh-add -d ~/.ssh/github_rsa
Identity removed: /Users/jiangxin/.ssh/github_rsa (/Users/jiangxin/.ssh/github_rsa.pub)
```

6.3.2. hub

对于命令行用户，GitHub提供了名为hub的命令行工具，对Git进行了简单的封装。该项目在GitHub上的地址为： https://github.com/defunkt/hub 。

使用hub可以在命令行中简化对GitHub的操作。例如克隆本电子书的版本库，若用hub命令，地址可大大简化：

```
$ hub clone gotgit/gotgithub
```

若要在自己账号下创建派生项目，无需登录GitHub网站，直接通过命令行即可实现：

```
$ cd gotgithub
$ hub fork
```

安装hub很简单，可使用如下方法任意一种方法。

- 克隆hub的版本库，从源码安装。安装步骤如下：

```
$ git clone git://github.com/defunkt/hub.git
$ cd hub
$ rake install prefix=/usr/local
```

- 用 RubyGems 包方式安装。

  hub用 Ruby 开发，也可用 RubyGems 包方式安装。需要注意，在安装完毕后最好将hub打包为一独立运行脚本，以便运行时不再靠 RubyGems 加载，提高加载速度。安装步骤如下：

```
$ gem install hub
$ hub hub standalone > ~/bin/hub && chmod 755 ~/bin/hub
```

安装完毕后，还需要对hub进行设置。定义两个Git风格的配置变量，以便hub命令能确定当前GitHub用户账号，并能够完成所需的 GitHub API 认证。

```
$ git config --global github.user "your-github-username"
$ git config --global github.token "your-github-token"
```

其中github.token中保存的是用户的API TOKEN，这在“2.1 创建GitHub账号”一节有过介绍。

在使用hub过程中，如果要为区分哪些命令是git的，哪些是hub的，而不断在两个命令间切换显然太不方便了。hub 命令支持以系统别名git的方式运行，即设置hub的系统别名为git，然后只需执行git命令，这样无论是git本身的命令还是hub扩展的命令都可正常运行。但要注意要用系统提供的别名方式，而不能把hub脚本改名为git，因为hub只是简单地对Git进行封装，运行时仍依赖git命令。在 bash 环境下建立别名可运行如下命令：

```
$ alias git=hub
```

其他 shell 环境下如何建立系统别名呢？运行hub alias 命令查看相关 shell 环境下建立别名的方法。例如对于 csh：

```
$ hub alias csh
Run this in your shell to start using `hub` as `git`:
  alias git hub
```

下面介绍hub的常用命令，节选自hub的项目页[1]。示例使用了别名命令git调用，并把对应的原始的git命令写在命令的下面（用提示符>表示，方括号中是说明）。

- git create

  在GitHub上创建项目。

```
$ git create -d '项目表述'
> [ 在GitHub上创建版本库 ]
> git remote add origin git@github.com:YOUR_USER/CURRENT_REPO.git

$ git create recipes
> [ 在GitHub上创建版本库 ]
> git remote add origin git@github.com:YOUR_USER/recipes.git

$ git create sinatra/recipes
> [ 在组织账号 sinatra 下创建版本库 ]
> git remote add origin git@github.com:sinatra/recipes.git
```

- git clone

  克隆版本库可使用URL简写，即“用户名/版本库”格式地址会自动扩展为Git协议（只读）地址或SSH协议（可写）地址。

```
$ git clone schacon/ticgit
> git clone git://github.com/schacon/ticgit.git

$ git clone -p schacon/ticgit
> git clone git@github.com:schacon/ticgit.git

$ git clone resque
> git clone git@github.com/YOUR_USER/resque.git
```

- git fork

  在GitHub自己账号下建立派生项目。

```
$ git fork
> [ 先在GitHub 上建立派生项目 ]
> git remote add -f YOUR_USER git@github.com:YOUR_USER/CURRENT_REPO.git
```

- git pull-request

  打开编辑器输入标题和内容，然后在 GitHub 上创建 Pull Request。

- git remote add

  设置远程版本库。和git clone命令一样支持URL简写。

```
$ git remote add rtomayko
> git remote add rtomayko git://github.com/rtomayko/CURRENT_REPO.git

$ git remote add -p rtomayko
> git remote add rtomayko git@github.com:rtomayko/CURRENT_REPO.git

$ git remote add origin
> git remote add origin git://github.com/YOUR_USER/CURRENT_REPO.git
```

- git fetch

  获取他人同名版本库。自动建立远程版本库并获取提交。

```
$ git fetch mislav
> git remote add mislav git://github.com/mislav/REPO.git
> git fetch mislav

$ git fetch mislav,xoebus
> git remote add mislav ...
> git remote add xoebus ...
> git fetch --multiple mislav xoebus
```

- git cherry-pick

  获取远程提交，并拣选至本地版本库。

```
$ git cherry-pick http://github.com/mislav/REPO/commit/SHA
> git remote add -f mislav git://github.com/mislav/REPO.git
> git cherry-pick SHA
```

- git am, git apply

  获取 Pull Request，并应用于本地版本库。

```
$ git am https://github.com/defunkt/hub/pull/55
> curl https://github.com/defunkt/hub/pull/55.patch -o /tmp/55.patch
> git am /tmp/55.patch
```

- git browse

  打开浏览器访问相应的URL地址。

```
$ git browse
> open https://github.com/YOUR_USER/CURRENT_REPO

$ git browse -- commit/SHA
> open https://github.com/YOUR_USER/CURRENT_REPO/commit/SHA

$ git browse -- issues
> open https://github.com/YOUR_USER/CURRENT_REPO/issues

$ git browse resque
> open https://github.com/YOUR_USER/resque

$ git browse schacon/ticgit
> open https://github.com/schacon/ticgit

$ git browse schacon/ticgit commit/SHA
> open https://github.com/schacon/ticgit/commit/SHA
```

- git help hub

  查看hub命令的帮助。

6.3.3. iOS应用

GitHub还为iOS平台开发了应用，这样就可以在 iPhone、iPad 等苹果设备上实时跟踪GitHub上的项目了。在苹果AppStore上搜索GitHub公司的应用，可以找到GitHub Issues和GitHub Jobs等应用，如图6-14所示。



图6-14：iPhone上的issues应用

在iPhone中安装GitHub Issues应用，就可以随时查看所关注的GitHub项目的问题报告和Pull Request等，如图6-15所示。



图6-15：iPhone上的GitHub Issues应用

而GitHub Jobs应用则和即将要介绍的GitHub招聘网站有关，用于浏览招聘信息。

6.4. 其他

6.4.1. GitHub:Jobs

GitHub求职网站[1]，于2010年8月开通，提供求职招聘服务[2]。还记得在“第2.1节 创建GitHub账号”介绍的相关内容么？当用户在个人设置中对简历和求职状态进行设置和启用后，GitHub就会帮助用户寻找合适的工作机会，而工作机会就来自于GitHub的求职网站（如图6-16所示）。

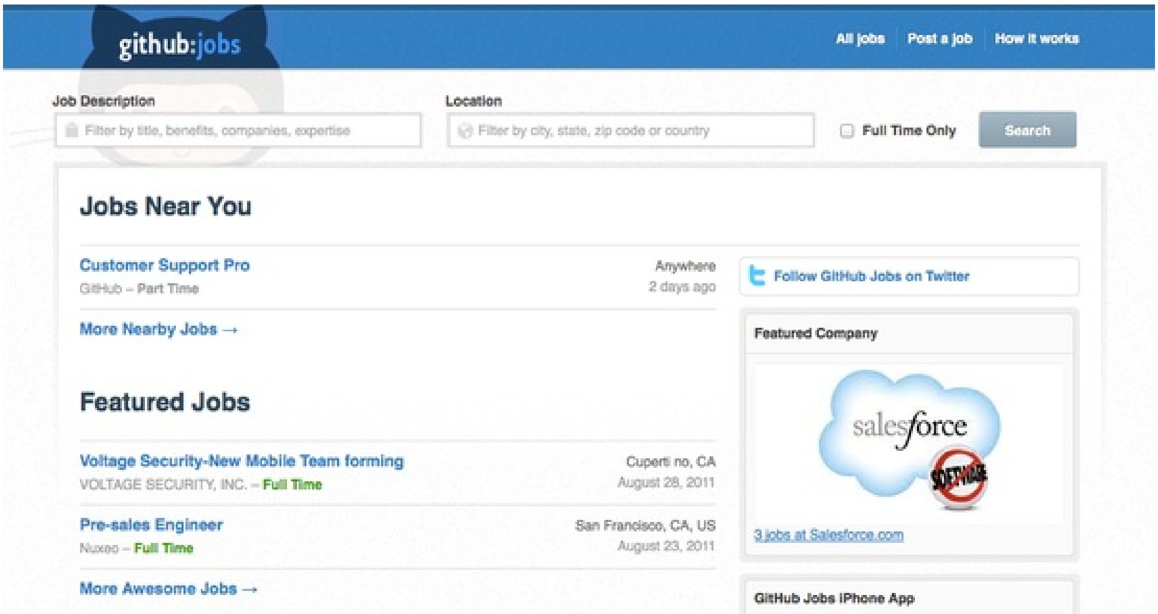


图6-16：GitHub求职网站

个人用户除了开启求职状态坐等通知外，还可以主动出击，直接到GitHub求职网站上寻找合适的工作机会，整个求职过程是免费的。而作为企业主发布招聘启示则是收费服务，发布招聘启示的流程如图6-17所示。

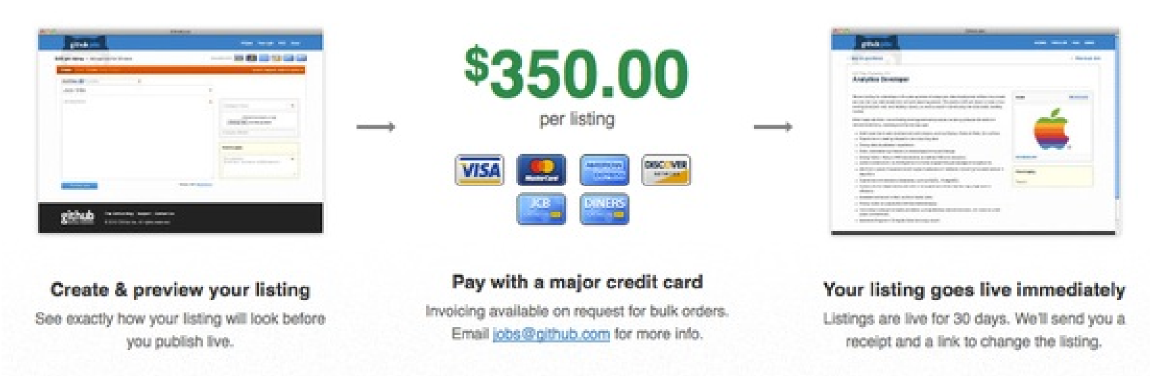


图6-17：企业主发布招聘启示流程

企业发布招聘启示，首先要按照模版填写职位说明及留下供求职者投递简历的邮件地址，然后用信用卡付费，每一个招聘启示的付费标准为350美元/月。一旦付费完成招聘马上生效。GitHub作为程序员的聚集地，无疑是招聘和应聘的理想之地。

6.4.2. GitHub:Shop

GitHub商店[1]销售着一些你在其他地方买不到的小东西——如印着GitHub吉祥物Octocat的纪念品，图6-18展示的就是一款印着Octocat的杯子。



图6-18：印着Octocat吉祥物的杯子

图6-19则是GitHub热卖的一款体恤的前后两面的图案设计[2]。体恤前面印着GitHub社区编程最核心的理念（fork you），体恤后面则可以用记号笔写下你在GitHub上的主页地址。



图6-19：超酷的GitHub体恤

GitHub商店实际上是架设于Shopify[3]电子商务网站上的网店。GitHub商店并非GitHub主业，销售纪念品可以增强GitHub用户的认同感，而GitHub粉丝可以购买一项“装备”让自己看起来更酷。

6.4.3. GitHub短网址服务

在“第2.2节 浏览托管项目”一节介绍图形文件差异比较时，需要给出一个网址，但这个网址很长。如下：

https://github.com/cameronmcefee/Image-Diff-View-Modes/commit/8e95f70c9c47168305970e91021072673d7cdad8

很自然地想到了Google短网址服务，于是由上面的长网址生成出一个短小精干的网址：http://goo.gl/Gy85b，访问该短网址会自动重定向到对应的长网址。

2011年11月，GitHub也推出了自己的短网址服务[1]，为GitHub自身网址提供短网址转换服务。GitHub短网址服务没有像Google短网址服务那样提供基于Web的图形化转换界面，而是需要用命令行进行网址转换。

例如对于网址 https://github.com/blog/985-git-io-github-url-shortener 的转换，使用curl命令如下操作。

- 将长网址转换为短网址。

  命令curl输出中的Location:语句即是转换后的短网址。

```
$ curl -i http://git.io -F 'url=https://github.com/blog/985-git-io-github-url-shortener'
...
HTTP/1.1 201 Created
...
Location: http://git.io/help
```

- 查看短网址对应的原网址，同样使用curl命令。

  命令curl输出302重定向地址即为原始网址。

```
$ curl -i http://git.io/help
HTTP/1.1 302 Found
...
Location: https://github.com/blog/985-git-io-github-url-shortener
```

为使转换的短网址更易于记忆和识别，可在curl命令中用 code 参数设定期望的短网址。例如下面命令将本节一开始提到的长网址转换为短网址：http://git.io/image-diff。

```
$ curl -i http://git.io -F \
  'url=https://github.com/cameronmcefee/Image-Diff-View-Modes/commit/8e95f70c9c4716830597
  -F 'code=image-diff'
...
HTTP/1.1 201 Created
...
Location: http://git.io/image-diff
```

6.4.4. GitHub Open Source

GitHub已成为新的开源项目大本营，而且GitHub也将其API开放，并将部分模块开源，借助社区的力量让GitHub变得更好。

GitHub大部分的开源项目托管在其官方账号下： https://github.com/github 。

API接口

GitHub通过域名api.github.com提供API接口，数据以JSON格式传递。

详细的API参考手册参见网址： http://developer.github.com/ 。API手册的版本库地址： https://github.com/github/developer.github.com 。

官方手册

GitHub官方手册参见 http://help.github.com/ ，使用 Jekyll 维护。

项目地址： https://github.com/github/help.github.com 。

Grit

Grit是Git的Ruby封装和实现，是GitHub调用Git的接口。部分是通过封装对git命令的调用实现的，部分则是纯Ruby实现。

项目地址： https://github.com/mojombo/grit 。

GitHub Services

Git版本库推送会触发服务器端post-receive钩子脚本。此项目将GitHub的服务器端钩子脚本开源，用户可以开发针对特定应用的钩子。GitHub还为其他GitHub应用提供了事件接口，如问题变更、Pull Request、维基页面修改等[1]。

项目地址： https://github.com/github/github-services 。

Hubot 和 Hubot Scripts

可以把 hubot[2]看做是GitHub的Siri（最早出现于iPhone 4S 的智能语音助理）或是新浪微博上的饮水姬[3]。GitHub将hobot和Campfire聊天室整合，hobot被聊天室会话触发可以实现诸如：打开办公室的门、根据wifi使用情况列出公司中的人、通过公司喇叭读一段信息等等许多好玩的事情[4]，而实现GitHub自动化部署则证明 hubot 可以完成更严肃的事情，在公司工作流中扮演举足轻重的地位[5]。

Hobot已经开源，项目库地址：https://github.com/github/hubot和 https://github.com/github/hubot-scripts（脚本）。

Gollum

GitHub以Git为后端的维基系统就是由Gollum实现的。每一个维基网页对应于一个文件，文件格式可以是 Markdown、textile、rdoc、org、creole、mediawiki、reStructuredText、asciidoc、pod 等。Gollum 调用名为github-markup的Ruby gem包（来自于下面要介绍的 Markup 项目）完成文件到网页的格式转换。

项目地址： https://github.com/github/gollum 。

关于GitHub维基参见本书“第4.6节维基”。

Jekyll

Jekyll 是一个简单的、支持博客的静态网站编译器。可以使用Markdown和Textile两种标记语言或者HTML撰写网页，并支持Liquid模版。实际上GitHub为托管项目生成静态网页使用的就是Jekyll。

项目地址： https://github.com/mojombo/jekyll 。

Linguist

Linguist 是一个Ruby模块，GitHub使用该模块对数据文件进行语义分析，检测文件的语言种类，代码加亮，对二进制文件进行忽略，限制非必须的差异显示，以及生成语言分类图等。

项目地址： https://github.com/github/linguist 。

Markup

GitHub通过这个ruby包对项目版本库根目录下的README文件，以及维基页面等文件进行解析、转换为网页显示。支持 Markdown、textile、rdoc、org、creole、mediawiki、reStructuredText、asciidoc、pod 等标记语言。实际上在对上述标记语言的解析和转换中，还依赖其他软件包，例如对于 Markdown 格式首选 Redcarpet(Redcarpet 是对一个高效的

Markdown解析器，通过对C语言的 Sundown 库封装实现。项目地址：https://github.com/tanoku/redcarpet。)，对 textile 格式使用 RedCloth，对 reStructuredText 格式调用外部命令rst2html，对 asciidoc 格式调用外部命令asciidoc等。

项目地址： https://github.com/github/markup 。

关于Markup软件包以及其他GitHub扩展的Markdown语法，参见：http://github.github.com/github-flavored-markdown。

Resque

Resque（发音类似 “rescue”）是一个以Redis为后端的Ruby包，用于创建和管理后台任务。可创建任务，将任务分配到多个队列，并在后台执行任务。

项目地址： https://github.com/defunkt/resque 。

GitPad

这是一个运行于Windows下类似Notepad.exe的应用程序，安装此应用后在Windows下做Git提交操作会调用类似记事本（Notepad）的应用撰写提交说明。

项目地址： https://github.com/github/GitPad 。

Maven Plugins

GitHub的Maven插件。

项目地址： https://github.com/github/maven-plugins 。

Gitignore

集合了针对各种语言环境的.gitignore（忽略文件）模版。例如其中针对VisualStudio的忽略文件模版Global/VisualStudio.gitignore部分内容如下：

```
# User-specific files
*.suo
*.user
*.sln.docstates

# Build results
[Dd]ebug/
[Rr]elease/
```

项目地址： https://github.com/github/gitignore 。

Media

提供GitHub网站Logo和吉祥物 Octocat 的图片，只能在授权范围内使用。

项目地址： https://github.com/github/media 。

7. 附录：轻量级标记语言

没有标记语言就没有Web和丰富多彩的互联网，但创造了Web的HTML语言并非尽善尽美，存在诸如难读、难写、难以向其他格式转换的问题。究其根源是因为HTML语言是一种“重”标记语言，对机器友好而并非对人友好。

下面这段HTML源码，非技术控阅读起来会遇到困难。

```
<html>
<head>
  <meta content='application/xhtml+xml;charset=utf-8' http-equiv='Content-type' />
  <title>轻量级标记语言</title>
</head>
<body>
  <h1 id='id1'>轻量级标记语言</h1>

  <p><strong>轻量级标记语言</strong> 是一种 <em>语法简单</em> 的标记语言。
  它使用易于理解的格式标记，没有古怪的 <code>&lt;标签&gt;</code> 。</p>

  <ul>
  <li>可以使用最简单的文本编辑器编辑。</li>
  <li>所见即所得，非技术控亦可直接阅读源码。</li>
  <li>可版本控制。</li>
  <li>实现单一源文件出版。</li>
  </ul>
<body>
</html>
```

同样的信息如果换用轻量级标记语言来表达，就非常直观了。如下所示：

```
轻量级标记语言
==============

**轻量级标记语言** 是一种 *语法简单* 的标记语言。
它使用易于理解的格式标记，没有古怪的 `` 。

- 可以使用最简单的文本编辑器编辑。
- 所见即所得，非技术控亦可直接阅读源码。
- 可版本控制。
- 实现单一源文件出版。
```

GitHub令人着迷的一个因素就在于GitHub为用户提供更为便捷地创建UGC（用户生成内容）的方法，其奥秘就在于使用了轻量级标记语言。无论是代码提交说明、提交评注、问题描述、项目的README文件、维基页面、用户主页和项目主页都可以使用Markdown(Markdown是在Ruby应用中广泛使用的标记语言，语法简洁并可混用HTML。标准的Markdown语法缺乏如表格等关键特性的支持，虽然不同的解析器都对其语法进行了扩展，但实现各有不同，造成一定的混乱。网址：http://daringfireball.net/projects/markdown/ 。等轻量级标记语言来撰写。轻量级标记语言如Markdown是对人友好的标记语言，一些语法参照了我们写电子邮件时的习惯，即使第一次接触用轻量级标记语言撰写的文件，也可以毫无障碍地理解其中的内容。

虽然GitHub更倾向于使用Markdown标记语言(GitHub使用Redcarpet作为Markdown的解析工具，并添加了额外的语法扩展。网址：http://github.github.com/github-flavored-markdown/，但很多地方也提供对其他轻量级标记语言的支持。包括为Python程序员所熟悉的reStructedText(reStructuredText可简写为reST或RST，是在Python中广泛使用的标记语言。reST的语法简洁严谨，本书就是使用Sphinx扩展的reST语法和工具撰写的。网址：http://docutils.sourceforge.net/rst.html，为Ruby程序员所熟悉的Textile(Textile是在Ruby应用中广泛使用的标记语言，例如Redmine就将Textile作为内置的标记语言。网址：http://redcloth.org/textile、RDoc(RDoc是内嵌于Ruby代码中用于维护软件文档的标记语言。网址：http://rdoc.sourceforge.net/doc/，为Perl程序员所熟悉的POD(POD是内嵌于Perl代码中用于维护软件文档的标记语言。网址：http://perldoc.perl.org/perlpod.html，为Emacs用户所熟悉的Org-mode(Org-mode是Emacs的一种编辑模式，除文档外还被广泛应用于维护TODO列表、项目计划等。网址：http://orgmode.org/org.html，为维基用户所熟悉的MediaWiki(MediaWiki是著名的维基百科（WikiPedia）所使用的维基语言。网址：http://www.mediawiki.org/wiki/Help:Formatting和Creole(维基的实现有上百种，语法各不相同。Creole试图建立统一的维基语法标准。网址：http://www.wikicreole.org/，以及可作为DocBook(DocBook是著名的用于文档撰写的标记语言，采用XML文件格式及大量的面向出版的格式标签，能够实现单一源文件出版（Single-Source Publishing），即一次撰写多种格式输出（Write once, publish many）。但复杂的XML标签给写作过程带来不小的负担。网址：http://www.docbook.org/前端的颇有前途的AsciiDoc(AsciiDoc的轻量级标签和DocBook的XML标签语法有着清晰的对应关系，既解决了DocBook语言标签复杂、难读难写的问题，又可利用DocBook丰富的工具链实现单一源文件向多种格式的输出转换。网址：http://www.methods.co.nz/asciidoc标记语言。

下面通过一张表格对几种常用的轻量级标记语言加以对照，供有不同标记语言偏好的用户参考，便于在GitHub某些不能随意更换标记语言而只能使用GFM（GitHub风格的Markdown）的场合可以自如地转换。

详细对照请参考原文

贡献者列表

Git和GitHub促进了开源软件的发展是因为消除了核心开发者和贡献者的隔阂——你若能看到代码，你就能改进代码。开放的电子书亦是如此，下面的贡献者让本书变得更好。

以贡献时间为序，感谢：

1. Zhang Hailong 报告文字错误。问题： #2 。
2. Riku 纠正文字错误。提交： 455d0db , f244e3d 。
3. windwiny 纠正文字错误。提交： 1ed1a51